DISTRIBUTED HANDLER ARCHITECTURE

Beytullah Yildiz

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University
July 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

_____
Prof. Geoffrey C. Fox (Principal Advisor)

_____
Prof. Dennis Gannon

_____
Prof. David Leake

_____
Prof. Andrew Lumsdaine

May 28, 2007

Acknowledgements

This dissertation has been achieved with the encouragement, support, and assistance I received from many remarkable people. I would like to offer my sincere gratitude to them.

First of all, I would like to thank my advisor Prof. Geoffrey C. Fox for his support, guidance and an exceptional research environment provided by him along the way of this endeavor. I deeply appreciate how much he contributed with his keen insight and extensive experience. His advice was always invaluable contribution to my academic life.

I would also like to thank the members of the research committee for generously offering time, support, guidance and good will throughout the preparation and review of this dissertation. I am very thankful to Prof. Dennis Gannon for his suggestions and remarkable inspiration, Prof. Andrew Lumsdaine for his constructive comments and Prof. David Leake for his rare combination of kindness and keen intellect.

I want to thank all former and current members of Community Grids Lab for the priceless moments that we shared together. I have had great pleasure of working with these wonderful people. I particularly thank Dr. Shrideep Pallickara and Dr. Marlon Pierce for their supports and productive discussions on various aspects of my research.

I was very fortunate to meet people who have become lifelong friends and cherished colleagues. My graduate studies would not have been the same without their invaluable contributions to my academic and social life. Dr. Mehmet Aktas, Dr. Galip Aydin, Dr. Hasan Bulut, Dr. Gurhan Gunduz and Dr. Ahmet Uyar did not only shape this dissertation with insightful comments and honest observations but my long and winding

road of graduate studies also became pleasing with them. I also owe Dr. Sangyoon Oh special thanks for many enlightening discussions and conversations. I feel very grateful to Dan Amonett who proofread this dissertation with an incredible patience and thoroughness.

Finally, I am especially grateful to my family and friends for their contributions throughout my graduate studies. I have received wonderful support from my parents, Riza and Fatma, my sister, Duriye and my brother, Emrullah. I am also thankful to my brother-in-law, Saban, and my sister-in-law, Emel. Their constant love and encouragement were invaluable. Their belief and generosity are most profoundly acknowledged here.

Abstract

Over the last couple of decades, distributed systems have been demonstrated an architectural evolvement based on models including client/server, multi-tier, distributed objects, messaging and peer-to-peer. One recent evolutionary step is Service Oriented Architecture (SOA), whose goal is to achieve loose-coupling among the interacting software applications for scalability and interoperability. The SOA model is engendered in Web Services, which provide software platforms to build applications as services and to create seamless and loosely-coupled interactions. Web Services utilize supportive functionalities such as security, reliability, monitoring, logging and so forth. These functionalities are typically provisioned as handlers, which incrementally add new capabilities to the services by building an execution chain. Even though handlers are very important to the service, the way of utilization is very crucial to attain the potential benefits. Every attempt to support a service with an additive functionality increases the chance of having an overwhelmingly crowded chain: this makes Web Service fat. Moreover, a handler may become a bottleneck because of having a comparably higher processing time.

In this dissertation, we present Distributed Handler Architecture (DHArch) to provide an efficient, scalable and modular architecture to manage the execution of the handlers. The system distributes the handlers by utilizing a Message Oriented Middleware and orchestrates their execution in an efficient fashion. We also present an empirical evaluation of the system to demonstrate the suitability of this architecture to cope with the issues that exist in the conventional Web Service handler structures.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The computing environment has demonstrated an architectural evolution based on models including client/server, multi-tier, peer-to-peer and a variety of distributed systems. One recent evolutionary step is Service Oriented Architecture (SOA) whose goal is to achieve loose coupling among the interacting software applications for scalability and interoperability.

SOA manifests itself perfectly in Web Service Architecture, supplying software platforms to build applications as services. Web Service Framework offers standard ways to interoperate among software applications, running on a variety of platforms [1]. It provides seamless and loosely coupled communications; applications can communicate with each other without much effort even though they might be utilizing different languages and platforms.

Why is interoperability so important? We can see its importance in our daily life. When we travel abroad we bring our laptop, cellular phone and our shaving kit. As we know, they need to be charged once they are used a certain amount of time. If a shopping center has not been visited to buy a converter, we would have wasted our energy by carrying those items because they became useless as a result of an incompatible plug.

If the world has a common standardized plug type, we would not have any problem when the devices need to be charged. Similarly, Web Service requires a common ground to offer interoperability. Hence, W3C defines Web Service to provide guidance:

> *"A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols."*

Many specifications have been introduced and many of them are on the way. The key features of the Web Services described by W3C have been introduced as Web Service specifications; Simple Object Access protocol (SOAP)[2], Web Service Description Language (WSDL) [3], and Universal Description Discovery and Integration (UDDI) [4] are the de-facto standards for Web Service Framework.

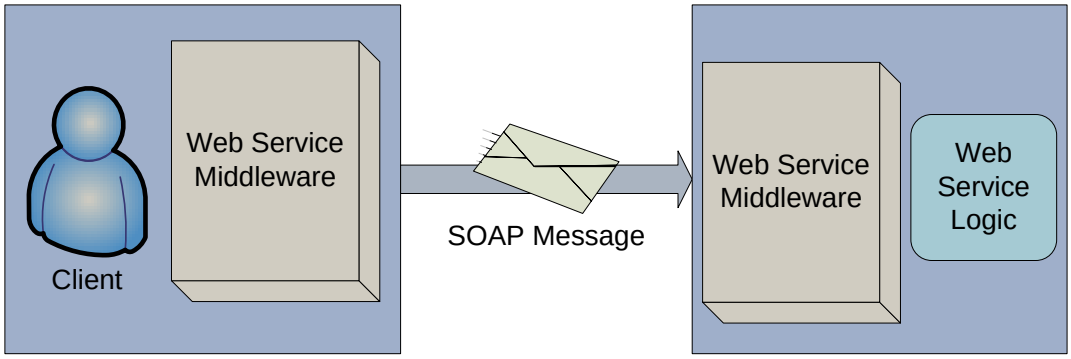


Figure 1-1 : A Simple Web Service Interaction

When we want to send an item by mail, we may request additional features from the post office. We prefer delivery confirmation for an important document. If the item is valuable, it is better to be insured. The post office basically delivers what we send. However, it offers additional capabilities to serve us better. Web Service Framework resembles it in many ways. It is simply a delivery of SOAP message.

One of the most crucial aspects of Web Service Framework is the utilization of the XML messaging. SOAP is an XML based data exchange format. The applications communicate by SOAP messages. Consequently, Web Service Framework heavily depends on SOAP processing. As a result, several Web Service containers, the middleware in Figure 1-1, has been introduced to take pressure off the applications. Their main goal is to hide the details of the SOAP processing from the users. The most popular containers are Apache Axis[5], Microsoft Web Service Enhancement[6] and IBM Websphere[7] .

The container architecture employs two main SOAP processing components, Web Service logic and handler. Handler is also called as filter. Web Service logic carries out the main task; it is a standalone application that is able to provide a service. On the other

hand, a handler is a supportive application. It contributes to a service with additional capabilities; such as reliability, security and logging.

Handlers offer new capabilities to services without increasing the complexity. Simplicity is a very crucial feature of applications. Handlers help to create Web Service so that the service acquires additional capabilities without touching the service implementation. Simplicity originates from very well known notion, *divide and conquer*. The whole task is divided between handlers and the service endpoint. Instead of having a large, hardly manageable application, clearly separable smaller tasks are more plausible. This notion contributes to have a simpler, efficient and modular structure.

Despite the fact that handlers preferably deal with the header, they also have the ability to modify the SOAP body. Many WS-Specifications have been introduced so far. They are the efforts where the community sets the standards to have more interoperable systems[8]. Some of them are very good candidates to be handlers, especially, those dealing with the headers. On the other hand, these specifications do not necessarily work with only the header. Many of them also affect the SOAP body. More detailed information about the effect of WS specifications on the SOAP parts can be found in Appendix D.

Web Services are able to employ a set of handlers to acquire many capabilities in a single execution. For instance, a service may need to be reliable as well as secure at the same time. Handler chains or pipelines are introduced for this purpose. Conventional Web Service containers contribute to the pipelining of the handlers by providing their own architecture. The container engine lets a message travel through the pipeline. Each handler receives processes and returns the messages in an order.

Consequently, the handler concept makes Web Service Architecture more modular rich and efficient. Instead of having a hardly manageable big chunk of application containing both service logic and its necessary functionalities, separating the functionalities from the service logic architecturally sounds better. This design allows Web Services to acquire incrementally additive functionalities without touching the endpoint. Thus, Web Service acquires unlimited richness in terms of the capability and features.

1.1 Motivation

Apparently, handler is a crucial aspect of Web Service Architecture because of the key importance in the execution path. However, the way of utilizing handlers and their structures become important when the number of the necessary additive functionalities increases. The efficiency becomes essential when power hungry and time consuming applications are introduced in the execution pipeline as handlers. For instance, reliability adds significant amount of processing time. Similarly, security necessitates powerful machines to conclude its task in a reasonable duration. Any additional handler may make the response time of the service soar. Services exhibit a many-to-one feature; many clients may ask many requests from a single service. Thus, the service side is influenced more dramatically than the client side.

Nevertheless, a service cannot be banned from obtaining new features. It is predestined that services will necessitate new capabilities to present a better computing environment. In other words, services eventually attain more handlers in their execution paths. Accordingly, we may wind up with an overwhelmingly crowded pipeline of the handlers. This circumstance will trigger that services become slower. This situation is

named as a Web Service becomes *fat*; while the service is acquiring new capabilities, the response time becomes longer and the management of the service becomes harder.

Secondly, a handler may cause a *convoy effect*. In an execution pipeline, a handler may delay the service processing due to the fact that its execution is too slow. In other words, a handler becomes *bottleneck*. This condition mounts the request messages waiting to be served in every second. The clients start waiting longer and longer.

Let's think about a highway, which has three lanes. And it is rush hour. Everybody is driving to reach home and get relaxed as soon as possible. However, at some point, the road becomes narrow; it operates with two lanes. Since, it is peak time; the road capacity is not sufficient to serve the arriving cars. In every passing minute, the number of cars grows. The people start becoming distressed because they do not want to waste their time in the highway by just waiting. How can we solve the problem?

The first solution is to expand the narrow part of the road. Adding one lane to the narrow part will suffice. The second solution is to detour a portion of the traffic to a parallel road. We can utilize both approaches in the handler architecture. Replacing the narrow road resembles introducing new enhanced computing environments. Using the parallel road looks like offering concurrent execution for the handlers.

We have additional resources out there. Networks are becoming faster. Machines are becoming more powerful and their speed is constantly improving. Hence, these improvements can contribute to remove inefficient parts. A single machine may not be enough. Bottlenecks can be eradicated by delivering handlers to the powerful computers. The distribution reduces the burden over a single computer.

Application parallelism is not new idea; it has been utilized for decades. Hence, handlers can be executed concurrently. However, handler parallelism is not utilized in the conventional Web Service Architecture. The parallelism boosts the performance and provides very effective and powerful solution.

It is being observed that many new technologies are being introduced in every day. Recently, an enhancement in processor technology becomes popular. Multi-core processors are started being widely utilized; even personal computers leverages cores offering opportunity for parallel executions. This opportunity contributes to the parallel handler execution even without introducing any network latency.

Distribution of the applications is very crucial to improve performance and scalability. However, there are requirements to be able to benefit from it. The decision of a handler distribution is influential over the system performance. Moreover, the selection of the handlers running concurrently is very vital. The conditions and requirements of the distribution are necessarily needed to be investigated extensively.

Handler structure demands efficient handler orchestration because there are many applications to be managed. The handlers have to be orchestrated in a way that Web Service benefits most. The orchestration is especially essential when the handlers are distributed. It becomes inevitable, when the concurrency is launched for the handler executions.

Reusability is one of the key features for an application. Instead of deploying the same handler many times, we may make use of the handler repeatedly. There are many stateless handlers. They process a SOAP message and return the results without keeping information. For instance, compression and decompression are stateless applications.

Hence, they are very suitable to be used by the services and/or clients many times without complications. Even stateful handlers may become appropriate to be utilized repeatedly in certain conditions.

> *"There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*
>
> -- *C. A. R. Hoare, The Emperor's Old Clothes*

Charles Antony Richard Hoare states a very essential feature to design excellent software [9]. Having a clean separation between the components to build so simple software that there are obviously no deficiencies is the goal. Simplicity contributes to constructing modular and flexible applications. However, it is a challenging effort to build a perfectly flexible and modular system. So it is for the handlers.

Interoperability is one of the most important features of Web Services. It assists to build seamless communication among the applications. The messaging is the key aspect behind the interoperability because it decouples the components from each other. We questioned ourselves about why we cannot leverage the messaging for the handler execution too.

Consequently, Handler Architectures need to be investigated to provide efficient, scalable and flexible Web Services. Since a SOAP task either related with the body or header may be costly, we need additional resources and structures. We can improve the performance, make the system scalable and provide improved architectures.

1.2 Statement of problems

Web Service Framework has very promising architecture offering an interoperable environment for distributed applications. One of the most crucial computing components of the services is the handler. Its execution has to be efficient, and effective. There are issues preventing this essential part of Web Services to gain its full capacity.

First of all, Web Services are increasingly becoming popular. They are utilized in the range from very simple application to very computation centric software systems. The computing power of the machines almost doubles every year following the projection of Moore's law[10], the network speed also catches up. Hence, the obtainable computing power increases steadily. Moreover, many other resources became accessible such as application software, storage, and sensor and so on. On the other hand, a conventional handler pipeline exploits a single machine. For this reason, we may hit a barrier if the handlers get complicated or more than a machine can handle. Web services are getting complex and requiring new features. Utilizing only one computing node prevents the services from taking off.

Secondly, the conventional handler structures are sequential. A chain of handlers processes the messages by passing them to each handler in an order. When a handler accomplishes its task, the next one receives it. However, there are many handlers that can be processed concurrently. The conventional handler architecture does not exploit this opportunity.

Thirdly, handlers generally are reusable applications such as security, logging, and monitoring and so on. Instead of deploying an instance of the same handler for every

service, the usage of already deployed handlers is more reasonable. The conventional handler architecture can utilize only the handlers deployed locally for a service because of the handler chain structure.

Fourthly, conventional handler mechanisms provide static handler structure. When handlers are deployed, the execution sequence cannot be change on the fly. However, XML based computation allows us many opportunities such as context based processing; there are sufficient tools to improve the handler processing sequence. The mechanism needs an ability to adapt according to the changing conditions.

Finally, utilizing many handlers requires an efficient orchestration. The execution order of handlers is very important. The selection of the concurrent pairs and the sequence is vital for efficiency. There are many options, but not all of them offer the same benefits. Some might result in disastrous consequences.

1.3 Why Distributed Handler Architecture (DHArch)

Distribution is the key feature to utilize additional resources, either hardware or software. In a single memory space, available resources may not be possibly accessed; many of them are out there and reachable via suitable means. DHArch offers an environment to utilize additional resources by distributing the handlers. It breaks the boundaries that keep the handlers in a single memory space that locks the handlers as if they are in a cage.

Even in a simple application, there are many tasks executed concurrently. A computer game may contain hundreds of parallel executions. Why cannot we use parallelism for the handler structure? Apparently, concurrency boosts the performance

and reduces the response time. DHArch offers an environment to benefit from the concurrency.

Since DHArch is able to distribute handlers into the different spaces, it improves the reusability. Handlers are deployed to well known addressable places that can be reached by many services and clients. They typically perform generic tasks such as security, monitoring, compression and so on. It does not matter who is requesting, they correctly perform the execution. Hence, it is very appropriate for a handler being used by many clients and services. DHArch is perfectly capable of achieving this mission.

Handler Distribution allows the replication of a handler. A handler can be duplicated and deployed to an addressable place. This notion can be utilized when a single handler cannot answer the request. DHArch is capable of providing a handler replication. The identical handlers can perform their own task independently in the DHarch handler execution environment.

A conventional handler execution mechanism employs a service specific handler sequence. In contrast, DHArch utilizes an individual handler execution sequence for every message. This attribute grants flexibility that every message may have its specific set of handlers. Moreover, this sequence can be modified on the fly. This characteristic of the DHArch contributes to adapt the execution according to the changing conditions.

A Web Service may have many handlers. Orchestration is the key feature of having the efficient distributed handler execution. Therefore, we introduced separation of description from the execution for the handler orchestration. It contributes to have efficient and effective handler structure.

Web Services, which exploit messaging, construct loosely coupled systems to enhance interoperability between applications. Similarly, handlers can profit from the messaging because it is innate for Web Service Framework. DHArch utilizes a Message Oriented Middleware (MOM)[11] for this purpose. MOMs are matured enough so that they guarantee message delivery. They offer asynchronous messaging. Additionally, using MOM for the internal messaging brings many benefits as by-products such as reliable and efficient delivery.

DHArch improves the scalability. Utilizing additional resources and introducing parallelism contributes to the handler execution. The usage of powerful machines or the distribution of the tasks among multi-core, multi processor or multiple machines causes the system to scale very well. Additionally, introduction of parallelism boosts the system performance. The throughput of a Service increases significantly.

1.4 Design features

Handlers are needed to build rich, modular, efficient and user friendly Web Service architectures. However, the way of using them is very critical. We contribute to the modularity, interoperability and responsiveness of the system by introducing a distributed approach for the handler architecture. A mechanism employing handler distribution is an outstanding solution. Disseminating the handlers among individual physical and/or virtual machines contributes to build more efficient, scalable and flexible structures.

Distributed Handler Architecture (DHArch) is basically a handler processing engine. In the long run, it may become a Distributed Operating System for the Web Services, Distributed Web Service Container. However, the focus of this dissertation is

specifically the handler architecture. Hence, DHArch basically offers an efficient distributed environment for the handler executions.

DHArch has many features and capabilities. It has the ability to work with different SOAP processing engines. It is able to offer its capabilities to the Web Service containers. In other words, it can cooperate with other SOAP processing engines by exploiting a gateway. Gateway is an interface between DHArch and native SOAP processing engine.

DHArch achieves handler execution with its specific structures. Therefore, it is able to autonomously process the handlers. It does not necessarily need other systems. On the other hand, it makes the common handler interfaces available to offer flexibility for the deployment. This characteristic prevents the compulsory modification on the currently implemented handlers. For instance, Axis handler abstract class, *BasicHandler*, works perfectly within the DHArch. Therefore, an Axis based handler can be processed in DHArch without a modification.

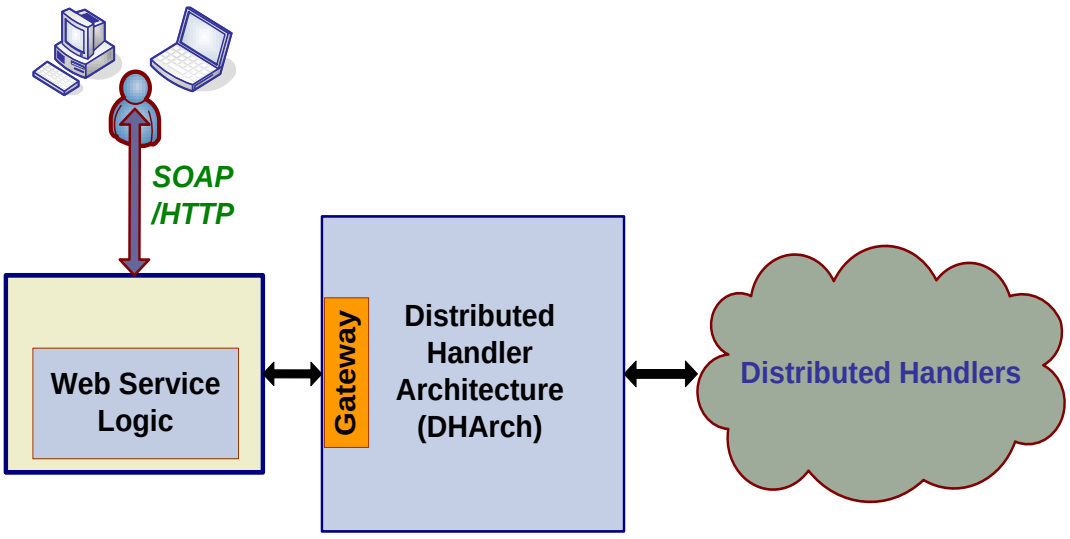


Figure 1-2: A simplified architecture of DHArch

DHArch is transparent to the users; the very vital feature of a Web Service interaction is not ignored while new capabilities are being introduced to Web Service Framework. The client is only aware of the address of the endpoint and the service definition; DHArch is not noticeable by the client. There is not any obvious distinction between conventional and DHArch utilized service usage in terms of the complexity of the accessibility.

Web Service Framework uses various transportation mechanisms and protocols. The Hypertext Transfer Protocol (HTTP) is the one mostly utilized. It is an application level generic stateless protocol for the distributed collaborative hypermedia information systems[12]. It provides a very suitable communication environment among the organizations; by utilizing HTTP, web services can work through many common firewall security measures among the different organizations and platforms without changing any firewall policies. However, it has also some limitations especially because of the request/respond paradigm. The request has to be followed with a response in HTTP. This results in unnecessary network usage for some cases. It does not support asynchronous messaging very well. It requires an upper level mechanism to handle asynchronous communications.

Consequently, in DHArch, messaging is utilized. A Message Oriented Middleware provides necessary tools for communication purposes. MOMs are mature enough so that they guarantee message delivery to specified addresses. NaradaBrokering[13] suits very well as a MOM that provides the necessary capabilities for the handler distribution. It acts as a post office that carries the messages between handlers and finally to the service end-point. It is able to provide reliable and secure

communication means to carry out critical tasks. It can also keep the messages until being delivered. Depending on the size of the message, thousands of messages can be queued to regulate the message flow.

In addition to the MOM utilization, DHArch uses its own structures to carry out the message execution. A context structure, Distributed Handler Message Context (DHMContext), assists to the distributed handler execution. Every request arriving to a Web Service endpoint is wrapped within this context. The flow structure, message and many other parameters are kept in the context to provide a convenient way of accessing to the required information during the execution.

The message is not the only necessary entity being passed to a distributed handler. Handlers may require further information. Therefore, an XML based message format, DHArch Messaging Format (DMFormat), is created to carry the supplementary data. It basically corresponds to the Message Context objects of the conventional Web Service containers.

Many requests may overwhelmingly arrive to a service and cause it to drop requests. Conventional containers accept request messages if there are available threads to work for them. Otherwise, the requests are rejected. DHArch provides an improvement by introducing a queuing mechanism. It employs the queues to keep the necessary information and to regulate the message flow; they improve the throughput by accepting and storing the request during the peak times and processing them when the influx reduces.

DHArch employs an individual processing engine. The engine contains many processes to accomplish the handler execution in a distributed fashion. The messages are

stored and selected to be executed according to the queuing scheme. Since each message contains its own handler sequence, the engine controls whether the sequence is correctly executed. The responses are also kept track by the engine. When a message response is received, the corresponding message context is updated with the guidance of the engine.

For the handler sequence, DHArch utilizes its own orchestration mechanism. Orchestration is very essential to perform the operations correctly and efficiently. The requirement of the orchestration became inevitable when some of the handlers are working concurrently. DHArch introduces a mechanism separating the description from the execution in order to have efficient and effective execution. By doing so, the complexity of the engine is being able to be reduced while the orchestration description is providing very powerful expressiveness for the handler sequences. Without sacrificing efficiency, acquiring simplicity was very challenging. However, the separation helps to succeed in this goal.

Every handler is hosted by a Handler Execution Manager (HEManager), the distributed portion of DHArch. Without having a supportive environment, handlers cannot perform their tasks in the distributed fashion. They need to be assisted. HEManager is considered to build a suitable environment. Every distributed handler hires its own HEManager. The manager contributes to the handler execution in many ways; stretching out from negotiating with message delivery system to the creating necessary structures for a distributed handler.

1.5 Contributions

DHArch offers architecture for an efficient, scalable and modular distributed handler execution. It introduces new ideas for the Web Service handler processing.

Concurrency is not a new concept. Many software applications perform concurrent execution. The aim is basically to enhance the efficiency by processing the applications concurrently. A handler can also benefit from being processed alongside another handler. Hence, DHArch possesses an ability to provide an environment for the parallel handler execution in an efficient way.

Moreover, DHArch offers an atmosphere to utilize effectively additional resources. Multi-core computers provide very promising environment for the handler execution. Moreover, the internet era hands over enormous amount of resources. While the network speed gets faster, the distributed resources become more available. During the period of Kbit/s and even Mbit/s in the network, handling the whole execution in a single system might be plausible. However, we are looking Gbit/s range nowadays. Transferring the applications to geographically distributed computing nodes has become more feasible. Hence, handlers can be delivered to the places where the resources are available. This capability feeds Web Service computation power with the tremendous additional resources.

DHArch introduces several structures to improve the handler processing environment. Leveraging queuing for the Handler architecture is very crucial. Queuing regulates the message flow. The messages are kept in the queue during peak times and executed when the computing components became available. There exist two queuing mechanisms. The first one is the DHArch internal queuing structure, containing several queues. The second queue mechanism is offered by NaradaBrokering to regulate the message flow to the handlers.

Additionally, DHArch is able to update handler sequences on the fly. This capability is acquired by having a unique context structure for each message. The context contains the handler sequence. In contrast to the static approach, this sequence can be modified during the execution. Hence, in addition to utilization of an individual handler set with a specific sequence, the sequence can be updated when it is necessary.

Moreover, DHArch utilizes its unique orchestration module to support efficient and effective handler processing. The execution engine needs to be very efficient. Our motto is "*so simple that there are obviously no deficiencies*". The complexity of the engine must be reasonable. On the other hand, the description has to be able to express any handler sequence. Therefore, DHArch benefits from the separation of the description from the execution for the orchestration.

In order to investigate our architecture, we conducted extensive experiments. We provide detailed performance evaluation. Several systems and environments are utilized to reach a conclusion about the general characteristics of DHArch.

Finally, we have investigated Web Service containers such as Apache Axis and realized that they are a collection of handlers which are contributing SOAP processing. To do so, they are working in harmony. Although this dissertation targets user level handlers, this research provides seeds for the next generation Distributed Web Service Operating System.

1.6 Research questions

Web Services are the manifestation of Service Oriented Architecture (SOA), which offers interoperable environments for applications. Previously, many technologies have tried to support interoperability such as CORBA, RMI, COM, and DCOM and so

on. At some point, more must be done. New opportunities and requirements have brought new demands. People started looking for better solutions. Web Service Architecture has very promising features to answer the demands and requirements. It has two crucial computing components from user point of view: the service endpoint and handlers.

In this dissertation, we focus our attention to the handlers. We raise several questions in our minds. Is the conventional handler architecture enough? How can we improve the architecture? Why do we need to improve it?

While we are looking for the answers of these questions, we explore the following research questions:

- What does handler distribution require?
- What is the role of messaging? How can this very key supporter of an interoperable system be utilized?
- How can we provide efficient and effective handler orchestration?
- How does distributed handler execution happen?
- Performance wise, is handler distribution plausible?
- Is there any overhead for the distribution?
- Does the handler distribution scale very well?
- What are the criteria for distributing a handler? What are the architectural principles of the handler distribution?

We answer these questions in Chapter 7.

1.7 Methodology

To evaluate our architecture, we chose Apache Axis 1.x [5] and Apache Axis 2 [14] versions to deploy Web Services. Apache Tomcat [15] is used as a servlet container.

It is developed in an open and participatory environment. It implements Java Server Pages (JSP) and the servlet specifications from Sun Microsystem.

In order to use in measurements, we investigate the handlers. They can be classified in two categories; real handlers and the handlers created for the testing purpose. In the first category, there are specification implementations and general purpose handlers. Apache provides several WS-Specifications implementations, which is appropriate to be a handler such as WSS4J [16] for WS-Security[17], Sandesha[18] for WS-ReliableMessaging [19] and Apache WSRF[20] for WS-Resource Framework [21]. Additionally, Grid Community Lab at Indiana University successfully implemented WS-ReliableMessaging, WS-Reliability[22], WS-Notification[23] and WS-Eventing. Moreover, we created our own handlers. Some of these handlers are testing purposes and some others are generic capability handlers such as logging, monitoring, XML Converters, XML Modifiers, compressor/decompressor, and security related handlers.

We performed many experiments by utilizing several hardware configurations to figure out the behavior of Distributed Handler Architecture. DHArch is able to use multi-core and multiprocessor systems for additional computing resources. We also investigate the utilization of additional machines in LAN network.

J2SE 1.4.2 and J2SE 5.0 are utilized. Java is a platform independent object oriented programming language. Many Web Service technologies have opted Java as a programming language because of platform independence. Hence, we also select it to benefit from already developed technologies for Web Services.

1.8 Organization of dissertation

This dissertation contains seven chapters. We explain Web Service handler concept, the conventional architectures and the contributions we have provided. The research illuminates the path that the handler architecture will travel. The experience we gain warns the obstacles on the road and provides recommendations for those who want to go in this direction.

Summary of the dissertation is covered in CHAPTER 1. We briefly address why and how we have achieved this research. The Web Service handler notion is described at the beginning of the chapter. In the remainder, we provide the motivations behind this dissertation and architectural facts of the implementation. Finally, we raise questions about the issues that we investigate.

In CHAPTER 2, the related works and underlying technologies are explored. The conventional handler architectures are investigated in detail in the first half of the chapter. Moreover, a close project, from Extreme Lab at Indiana University, is examined because of importance of its contributions. Several technologies also find their place in the second half of the chapter because of their significance in this research.

The details of Distributed Handler Architecture (DHArch) are given in CHAPTER 3. DHArch has a modular structure so that it can be easily maintainable and improvable. At the beginning of the chapter, the big picture of architecture conveys the general idea and principles. The modules are explored in the remainder of the chapter.

The handler orchestration is very essential part of DHArch. We spare CHAPTER 4 to explain the details of the orchestration. Several work flow systems are investigated

briefly. We express the details of building constructs and the interpretation of orchestration documents during the execution in the remaining of the chapter.

The execution of Distributed Handler Architecture is discussed in CHAPTER 5. The message acceptance, distributed execution and receiving response cycle are explored in detail. The complementary features and capabilities are analyzed and necessary conclusions are driven in the last part of the chapter.

We collect the measurements in CHAPTER 6 and provide the detailed analysis for them. There are four experiment sub-sections. The first one contains the performance test results. They are gathered in various environments to figure out the behavior of DHArch. The second section investigates the overhead of the handler distribution in many environments. The third section provides the results for scalability. Lastly, we finalize the experiments by deploying the well-known WS-Specifications and gathering the performance results for the execution.

Finally we conclude our dissertation in CHAPTER 7. We provide a very brief summary and answer the questions which are raised in CHAPTER 1. Finally, we express our intention for the future researches.

CHAPTER 2

BACKGROUND AND SURVEY OF TECHNOLOGIES

2.1 Handler structures

We have investigated the related works that provide an execution environment for the Web Service handlers. We will discuss them in the remainder of this section.

2.1.1 JAX-RPC

JAX-RPC [24] provides interoperable Web Service environments across heterogeneous platforms and languages. It uses SOAP and WSDL as standards; WSDL defines Web Services. SOAP messages are utilized to transport the payloads. JAX-RPC offers necessary tools to ease the implementation and the usage of Web Services, shown in Figure 2-1. The tools help to hide the complexity of underlying runtime and SOAP

protocol level mechanisms such as marshalling and un-marshalling. Additionally, WSDL-to-Java and Java-to-WSDL create mappings for the client and service. It also utilizes SAAJ API which provides a library to construct and manipulate SOAP messages with attachments [25].

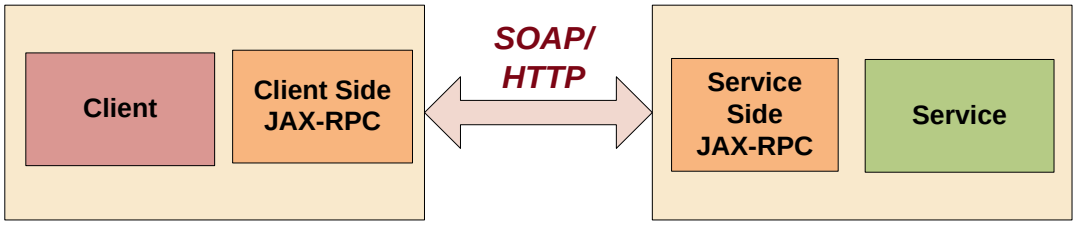


Figure 2-1 : JAX-RPC architecture

JAX-RPC makes use of HTTP as an underlying transportation protocol. Utilizing HTTP protocol provides additional capabilities such as HTTP level session management and SSL base security.

JAX-RPC specification offers a handler structure that provides an environment to add new capabilities to Web Service endpoints. It has javax.xml.rpc.handler.Handler interface containing three methods: handlerRequest, handleResponse and handleFault. Users can also write customized handlers by using Javax.xml.rpc.handler.GenericHandler abstract class.

A JAX-RPC handler is able to intercept SOAP messages at several points. This interception can occur in the client and/or service side. A client side handler is able to process SOAP messages before entering the network. A service side handler may interrupt these messages before they reaches to the endpoint. For example, a secure interaction requires a pair of handler. A security handler is added to the client side so that the client can authenticate itself and/or encrypt the messages. A counterpart of this handler in service side receives those messages and forwards them to the endpoint after

completing its security related tasks. Handlers are also able to be deployed to the response path. For instance, response messages can be monitored on the way through the client with a monitoring handler.

Several handlers can construct a handler chain in JAX-RPC. A chain may contain many handlers, whose executions are sequential. The execution order of the handlers needs to be defined while the service is being deployed. In other words, the deployment of the handlers is static; the execution path cannot be modified after being deployed.

2.1.2 Apache Axis

Apache Axis is currently the most dominant container in the Web Service community and has a plethora of applications developed around this container. There are two main versions, Apache Axis 1.x and Apache Axis 2.

2.1.2.1 Apache Axis 1.x

Axis 1.x is a Web Service container, which contributes to SOAP processing with many capabilities. It basically provides three main interfaces: Remote Procedure Calls (RPC) [26], document/wrapped and message style communications. In the RPC style, a Java object is serialized into XML and de-serialized back into a Java object at the target point. This is very useful if a Java program, which needs to be deployed, has been already implemented. Document and wrapped style are similar to each other, but differ in their use of SOAP encoding. The data is encapsulated within a plain XML document. Although serialization and de-serialization operations are not required, binding is needed in this type of deployment. The Message style is a user-defined style and is typically very

flexible. Since the message is already an XML document, serializers and deserializers are not needed.

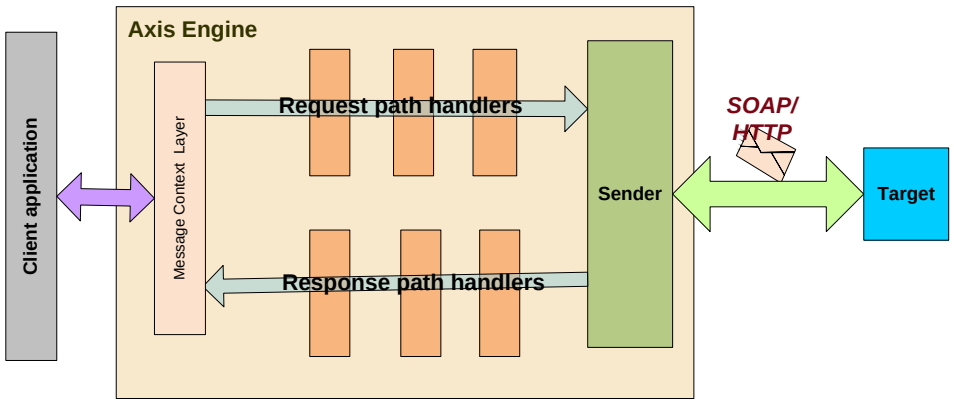


Figure 2-2 : Client side Apache Axis handler architecture

Similar to JAX-RPC, Apache Axis 1.x facilitates the incremental addition of capabilities by leveraging handlers. Handlers provide new features to the clients and services. Figure 2-2 depicts the client side architecture. The requests are intercepted by handlers before the network and the responses from services are captured first by them.

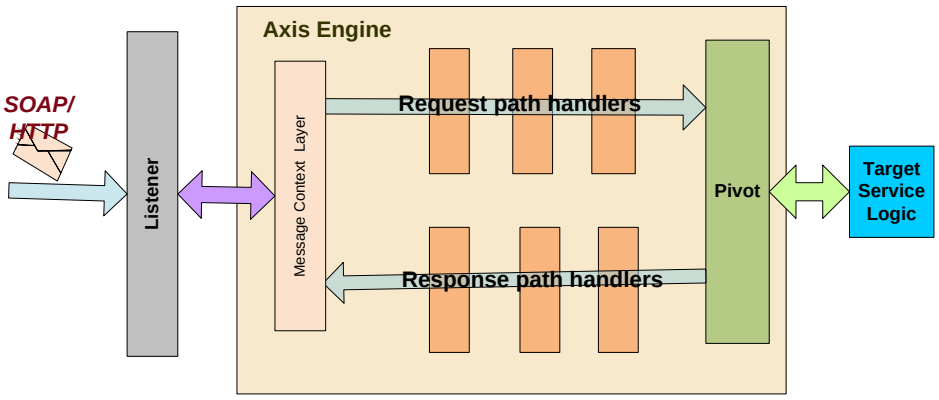


Figure 2-3 : Service side Apache Axis handler architecture

The architecture of the service side is illustrated in Figure 2-3. Handlers can be either request or response path. At one point, a handler sends request as well as receives response. This handler is called pivot handler. It processes requests and passes them to

the endpoint. When the endpoint finishes its tasks, the responses are sent back to the pivot handler.

There exist two types of handlers. The first type contains singleton handlers, which do not require a peer. They can be deployed to either client or service side. On the other hand, there are handlers that necessitate peers in the client and the service sides. This kind of handlers forms the second type. For instance, an encryption handler which encrypts messages coming from a client requires an inverse handler at the service side which performs the appropriate decryption. Client side handler peers are processed in the reverse order of the service side handler peers. For example, if a client processes handlers in the order of h1, h2 and h3, their counterparts in the service side are executed in the order of h3, h2 and h1.

When Apache Axis engine runs, it starts invoking a series of handler executions. Messages travel through a handler chain within a context, MessageContext. This context, which contains the message and additional information, is received by the handlers according to their positions in the execution chain. When a handler completes its execution, it passes the context back to the engine. The engine gives the context to the remaining handlers in the order that defines by the handler chain.

In Apache Axis 1.x, handlers can be transport-specific, service-specific or global. Hence, a message execution comprises three handler chains: transport, global and service. Custom handlers can be added to the Service-specific handlers.

We have investigated Apache Axis 1.x and given suggestions to make improvements. We have driven several conclusions that describe necessary features for a Web Service container [27].

2.1.2.2 Apache Axis 2

Apache Axis 2 provides a Web Service middleware that hides the complexity of SOAP processing from users. It has an extensible and modular architecture. The core modules are separable from the remaining modules so that the new modules can be added on the top of the core modules [28].

There are several core modules in Apache Axis 2. To handle information and keep the states, Apache Axis 2 defines an Information Module, depicted in Figure 2-4. Information module has a hierarchical structure that helps to manage the object lifecycles. It has two main hierarchies: contexts and description. The description hierarchy represents the static data that can be loaded from a configuration file. For example, *service.xml* stores the static information for Web Services. However, context hierarchy keeps the dynamic information that needs to be stored while execution continues. This structure is called a hierarchical structure because the context and description data are bounded with a key. When access to the data is required, an inquiry starts bottom to up. The lower level match is preferred to the upper level match. If data is not found in the lower level, it is searched in the upper level. This search continues to the top level. There is one down side of this approach; the search can take too long if the data, which is being searched, is not in the hierarchy[14].

Deployment module of Apache Axis 2 utilizes three level configuration structures: global, service and module. The corresponding configuration files are *axis2.xml, services.xml* and *module.xml*. When the container is started, the deployment module first creates the data structures by using global data of *axis2.xml*. Then,

module.xml is checked and finally *services.xml* is utilized to finalize Axis configuration. The static context structure is built on the top of these configurations.

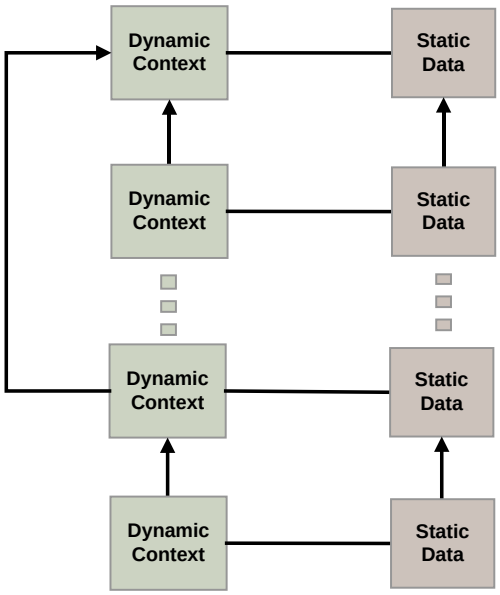


Figure 2-4 : Information Model

Since SOAP is the most important asset in the Web Service framework, the efficiency of the SOAP processing is very important for the overall performance. Therefore, Apache Axis 2 provides an efficient API, Axis Object Model (AXIOM). AXIOM is a lightweight XML info-set representation based on StAX (JSR 173) [29]. It is a standard streaming pull parser API. Contrary to the object model parser such as DOM, a pull parser does not create any object if it is not necessary. AXIOM utilizes caching; it allows creating and keeping objects for the pulled stream. Fortunately, this cashing can be turned off when it is not required to increase efficiency[30].

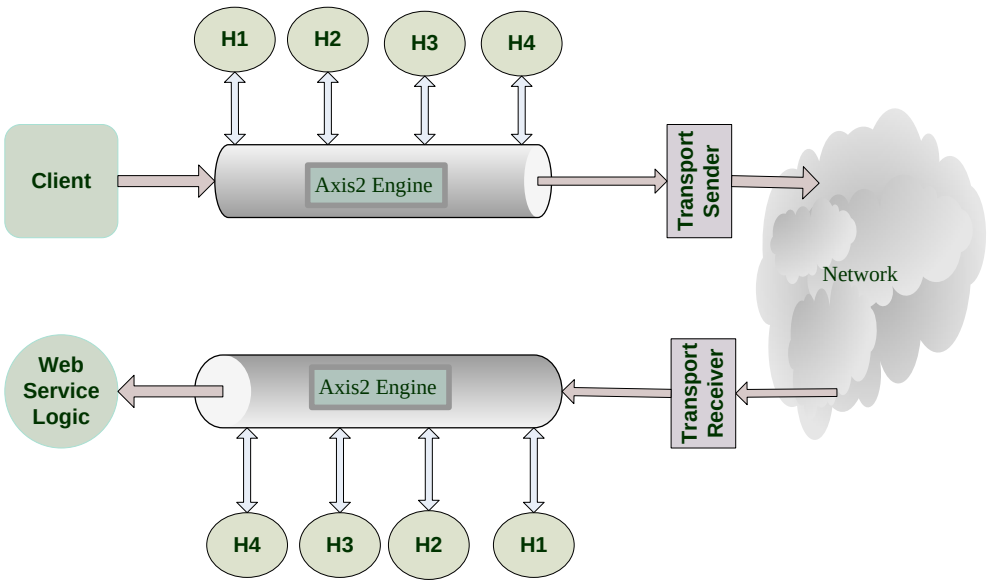


Figure 2-5 : SOAP Processing Model

There are two basic actions in order to process SOAP messages in Apache Axis 2: sending and receiving SOAP messages. Apache Axis 2 basically views every transaction as a single SOAP processing. To implement a complex SOAP messaging, containing several messages, a top layered framework is necessary. Apache Axis 2 framework contains two pipes: IN and OUT. They may be combined to exchange messages. Figure 2-5 shows the traversal of a SOAP message. User application can create a SOAP request by using a client API. Before handing the message over transport sender, new capabilities can be added with the handlers. They provide extensibility to the SOAP processing model. They can intercept messages in either IN or OUT pipe.

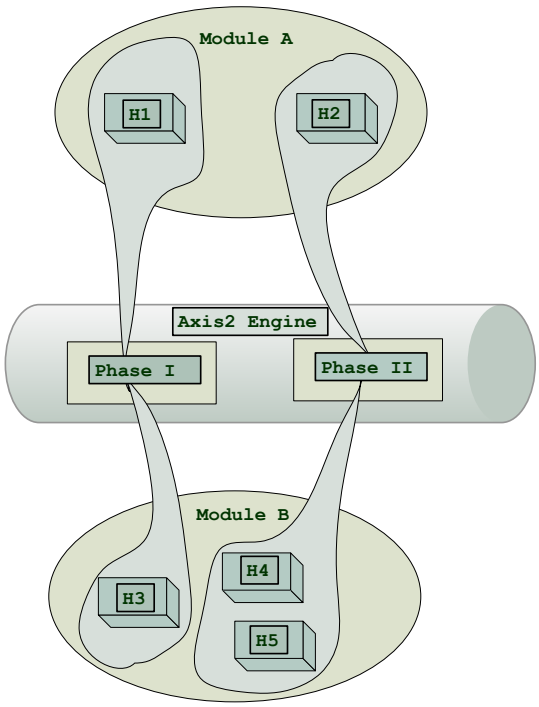


Figure 2-6 : Phase Module and Handler relation

Additionally, Apache Axis 2 introduces an upper level abstraction on to top of handler layer: module. A module may contain a set of handlers and phase rules, depicted in Figure 2-6. In other words, it groups a set of handlers to provide a specific functionality. They are basically intended to implement Web Service Specification in a modular manner such as WS-Addressing [31] and WS-Reliable Messaging[32].

There are stages to arrange the order of the modules. These stages are called as phases. Phases and flows together manage the processing flow for a specific message, depicted in Figure 2-7. Apache Axis 2 contains predefined special handlers such as Dispatchers, Transport receiver and Transport sender. Similarly, several predefined special phases are also introduced: Transport, Pre-Dispatch, Dispatch, User defined and Message Processing phases in IN pipe and Message Initialization, User and Transport phases in OUT pipe. However, this mechanism is not fixed; it is extensible and user customized phases and handlers are allowed to be attached.

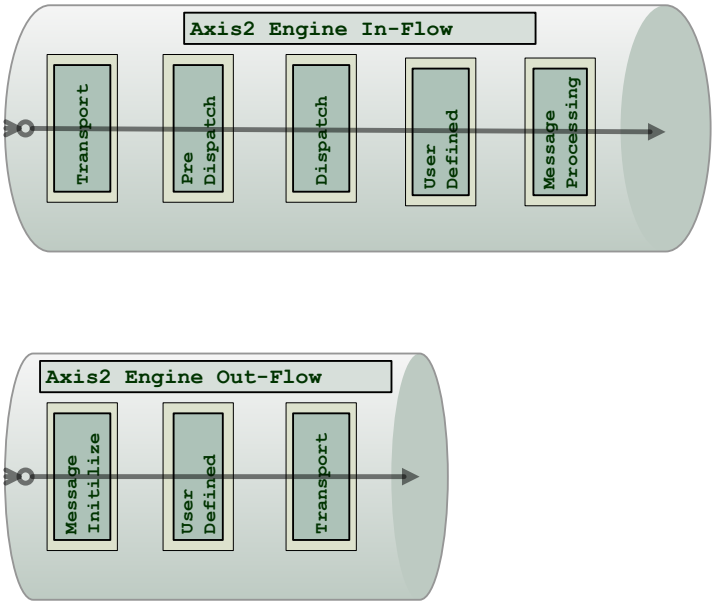


Figure 2-7 : Axis2 Engine In and Out Flows

2.1.3 Web Service Enhancement (WSE)

Similar to Apache Axis, WSE supports Web Services by offering an environment for the supportive capabilities, which are called *filters*. The execution structure of the filters is very similar to that in Apache Axis. The architecture is depicted in Figure 2-8.

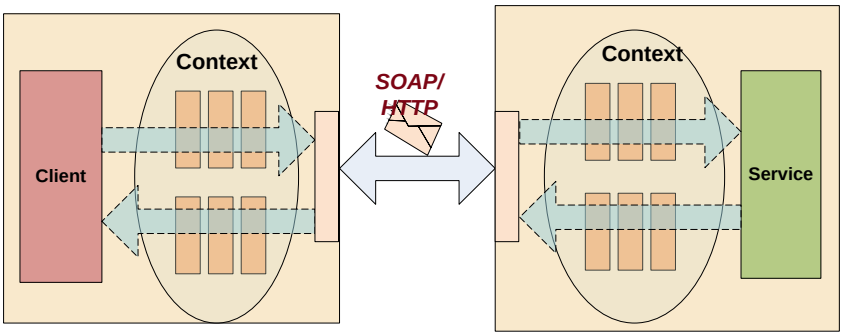


Figure 2-8 : Filter execution structure in WSE

Both Output and Input filters are capable of processing SOAP header and body. The real target is the header, though. WSE has already several build-in filters. However, customizable filters can be added. Filters can create a chain. In this chain, the intermediary information is passed between the filters by using a context, illustrated in Figure 2-8. The context provides an environment to share the properties and variables.

There is a difference in the message context structure between Apache Axis and WSE. MessageContext object in Apache Axis wraps a SOAP message and supplementary information. It is passed as a parameter through the handler chain. On the other hand, filters in WSE loads the information to the context object. The object does not contain the relevant SOAP message.

2.1.4 DEN and XSUL

XSUL is a modular Java Library to construct Web and Grid Services [33, 34]. It has been developed by Extreme Lab at Indiana University. It provides a framework for XML based processing and supports doc-literal, request-response and one-way messaging. Furthermore, it contains modules for a lightweight XML/HTTP invoker and processor. It also supports SOAP 1.1/1.2 and digital signatures. Moreover, it has an ability to provide dynamic service invocation.

DEN addresses the performance and scalability bottleneck [35]. It targets directly to the Web Service Security Processing steps without touching the Service logic at all. It granulates the application and makes the pieces separate processing nodes. These nodes are distributed across the Grids. It is able to execute these processing nodes concurrently. They must be independent entities to be executable parallel, though. The whole scenario is depicted in Figure 2-9.

The latest version of XSUL, XSUL2, allows a request goes through a chain of handlers until it reaches the destination. DEN by utilizing XSUL2 is able to separate the handlers from Web Service endpoint and distribute them as individual service nodes within a chain. The service nodes use a context to convey supplementary information for. A context stores intermediate processing results for the nodes. It is passed inside of the SOAP header through the handler chain.

Design of DEN is not restricted with only SOAP style Web Service interactions. It also utilizes generic HTTP commands such as GET. Moreover, XSUL2 offers an environment to benefit Web Service Definition Language (WSDL) by using GET command.

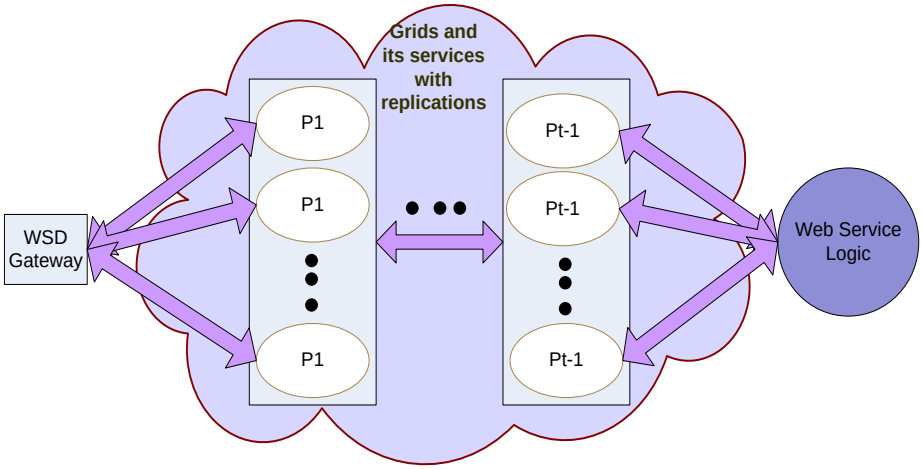


Figure 2-9 : DEN: WS Processing in Grids

Additionally, DEN utilizes a table, a hash table with registered service names as keys and endpoint vectors as values, for the routing purpose. Endpoint vectors contain at least one reference. These vectors are bounded with either intermediary processing nodes or final destination of the interaction.

DEN and XSUL is able to utilize asynchronous messaging by using WS-Dispatcher. The dispatcher allows internal services to be exposed to Internet. With the

support of asynchronous messaging, a WS-Security implementation is divided into sub-atomic tasks and deployed as services. Some tasks are executed in a parallel manner to gain performance and to remove the bottleneck.

2.1.4.1 Differences

The differences can be categorized into two: conceptual and architectural differences. DEN and XSUL distribute handlers as Web Services. Each handler utilizes a WSDL to broadcast information about how to be communicated. On the other hand, DHArch keeps the status of handlers unchanged; they architecturally remain as handlers. The distributed handlers in DHArch do not prefer utilizing a WSDL to be communicated even though it is possible to do so. DHArch prefers using the handler interfaces of the underlying containers and its customized handler interfaces. Hence, a currently implemented handler can be easily deployable without any modifications.

The conceptual differences let the architectures evolve into different directions. Since DEN and XSUL exploit Web Service Framework to deploy handlers, it makes use of HTTP protocol for communication purposes. The handlers are orchestrated by using a routing table. In contrast, DHArch utilizes a MOM to provide communication between the distributed handlers. On the top of this communication mechanism, an orchestration module organizes the distributed handler execution.

In short, even though some common issues originated from the utilization of the additive functionalities are targeted, the approaches differ conceptually and architecturally. Section 2.1.4 and CHAPTER 3 provide the architectural details of DEN+XSUL and DHArch respectively. We will look at the strong points and advantages of these approaches in the following section.

2.1.4.2 Strong points and advantages

DEN and XSUL together target directly to the Web Service security processing steps without touching the Service logic at all. They tear and granulate the security processing node and deploy the security tasks as individual services. This approach sets an example to distribute the handlers as Web Services.

Utilizing Web Service approach for the handlers provides several benefits. The first benefit is to be able to remove bottlenecks from the SOAP processing pipeline with a very well-known style. Additionally, service based approach improves the interoperability of the deployment. Moreover, this approach is able to utilize the tools that have been already implemented for the Web Services. For instance, many service level orchestration tools can be easily utilized. On the other hand, DHArch follows a different approach to provide a scalable, efficient and modular environment for the handlers. It is basically a distributed Web Service Handler Container. It also removes the bottlenecks from SOAP processing pipeline by using additional resources and providing an environment for the concurrent execution. Additionally, DHArch is able utilizes a Message Oriented Middleware (MOM) for the transportation purpose. MOM is a powerful tool to provide asynchronous, reliable, efficient delivery mechanism. In addition to excellent messaging capability, it can provide a queuing mechanism for the handler execution to regulate the message flow.

Furthermore, DHArch has its own handler orchestration mechanisms. The orchestration separates the description from the execution. This allows having a very simple and efficient execution structure while offering a very powerful expressiveness for the orchestration.

Moreover, DHArch provides a control mechanism for the pipelined message executions. The number of executable messages is kept in optimum level to prevent the performance degradation because of too many messages processing in a moment.

DHArch utilizes a context that allows a message based execution. Every message may have an individual handler execution sequence; DHArch may employ a different handler execution sequence for each message. It is not centralized. Instead, the sequence that is kept in the context may differ from message to message.

While we were deciding about the best way of carrying the data to the distributed handlers, we conclude that having a format which wraps the SOAP message and additional data is more efficient. The whole SOAP message does not have to be parsed in order to get necessary information to carry out the distributed execution. The format should be simple but not have any deficiency. Hence, the specific messaging format, agreed between the computing nodes, is created.

2.2 Technologies

Since one of the most important assets of Web Services is SOAP, an XML document, parsers are necessary in many places. Moreover, messaging middleware is one of the crucial components of DHArch. It has to provide necessary capability for the transportation. Because of their importance, we will explain these technologies in the remainder of this section.

2.2.1 XML Parsers

There are several parsers to utilize in XML processing. DOM Parser is the most widely used one. It reads and validates the XML document. If the document is valid, the

parser returns a document object tree, depicted in Figure 2-10. We can randomly access any element because each element is entirely kept in the memory. This capability provides very efficient navigation mechanism over the document. Hence, it is very suitable parser if the document needs to be accessed many times. On the other hand, it is a relatively process intensive parser and requires large amount of memory. This situation worsens if the XML document gets bigger. It does not allow partial parsing to remove this bottleneck.

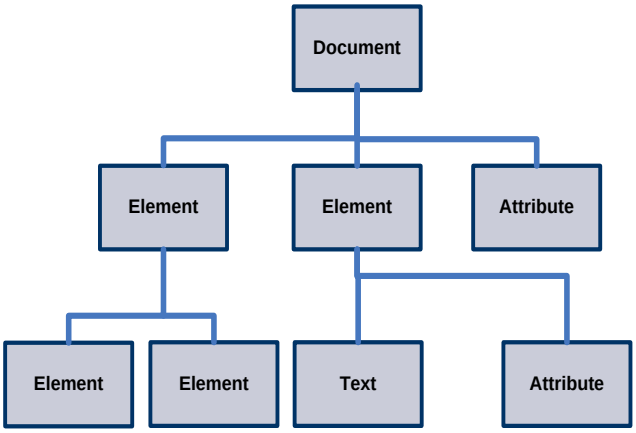


Figure 2-10 : A DOM Tree

On the other hand, SAX parser does not build a document object tree. Instead, it utilizes an event-driven push model. The parser reads a series of events from the XML document and pushes them to the event handlers. The context can be reached by using these event handlers, shown in Figure 2-11. SAX parser has low memory consumption because the whole document does not need to be loaded into the memory. SAX can handle a document whose size is bigger than the system memory. It also validates the XML document against XML schema. Moreover, it works faster than DOM parser. However, the document needs to be parsed repeatedly when the event states are not kept

for the later usage. Maintaining the events is left to the application. Therefore, it does not provide efficient XML parsing when the document needs to be accessed many times.

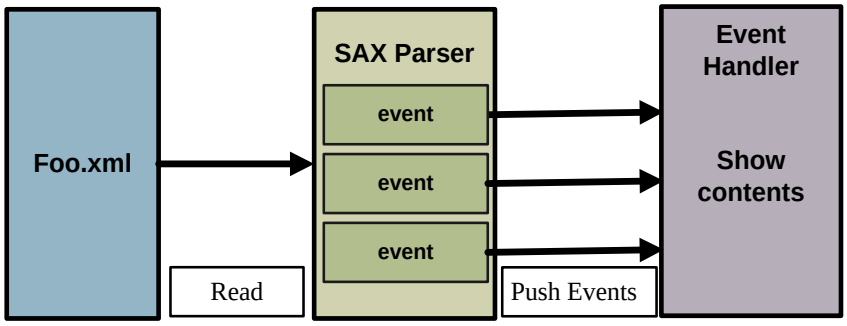


Figure 2-11 : SAX shows the context to an application as a series of events

StAX Parser provides a new parsing technique that utilizes an event driven model by pulling instead of pushing. It processes a request by returning the events as well as providing the relevant objects, depicted in Figure 2-12. StAX differs from DOM and SAX by specifying two parsing modules; curser an iterator module. Cursor module returns only events although iterator model provides objects. This allows creating an object if only it is necessary [36].

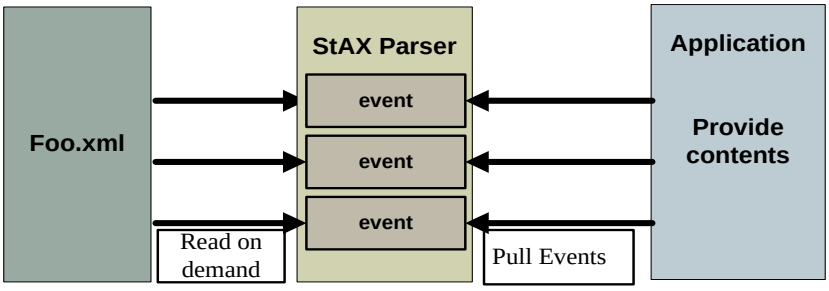


Figure 2-12 : StAX provides an event to an application

Performance wise, pull parser beats the previous parsers in most respects. For the limited memory environments pull parser is the most appropriate one because it requires very tiny memory space and can provides the object of the partial XML documents. If an

application does not require the validation of full document, entities, processing instructions or comments, pull parser looks to be best choice. The detailed performance results of various parser implementations are provided in an article, published by IBM [37]. XPP, an XML pull parser implementation by Extreme Lab at Indiana University, achieves very astonishing performance results[38].

2.2.2 NaradaBrokering

NaradaBrokering is a distributed brokering system that support centralized, distributed and peer-to-peer (P2P) interactions [39]. It have designed to operate on a large network of cooperating broker nodes, which are able to intelligently process and route messages while working with multiple underlying communication protocols. *Broker* is the smallest unit of the underlying messaging infrastructure. Broker nodes are organized in a cluster-based architecture that allows supporting large heterogeneous client configurations to scale in arbitrary size. NaradaBrokering imposes a hierarchical structure over the clusters. A broker is a part of cluster that is a part of super-cluster, which is also a part of super-super-cluster and so forth. These clusters consist of strongly connected brokers with multiple links to the brokers in the other clusters. This ensures alternative communication routes while a failure occurs. Every cluster unit employs a cluster controller node to provide a gateway to the other units

In order to optimize a routing destination, a broker node needs to be aware of the broker network layout. However, this is impractical when the potential size of the broker network is considered. Thus, NaradaBrokering utilizes Broker Network Map (BNM). BNM conveys information regarding the interconnections among the brokers of a cluster and the interconnections between the clusters in a super-cluster. A state change in a

network is propagated to only those brokers whose network is altered. The propagation of the connection information is not allowed to the outside of a cluster if the information regarding the connection is related to the brokers of the same cluster.

Event routing is very crucial task in NaradaBrokering. It includes matching the content, computing the destination and routing the content to the destination. When an event is sent from one node to others, individual unit controllers compute the best routes to those nodes that the event is needed to be delivered. At every node, the best decision is taken according to the current state of the network.

Matching engine computes the destinations associated with an event based on the profiles. The destination connected with a profile is added to the computed destination when the profile successfully matched to an event. NaradaBrokering contains five matching methods; string base matching, string based matching coupled with SQL-like queries on properties, topics that are based on tag=value pairs, integer based matching and XML based matching with XPath queries.

NaradaBrokering has an extensible transport framework. It supports multiple transport protocols such as TCP (blocking and non-blocking), UDP HTTP, SSL and RTP [40]. Moreover, since the channels between the interacting entities are virtualized, they can communicate across firewalls, proxies and NAT boundaries. Furthermore, NaradaBrokering has an ability to monitor the link states to measure loss rates, communication delays and jitters[41]. NaradaBrokering is utilized asynchronous communication. It can support different interactions by encapsulating them with specialized events.

One of the most important entities in the transport framework is *link* primitive. It encapsulates operations between the endpoints and abstracts details associated with the handshakes and communications. Link primitive has an ability to specify the changes of the underlying communication environment. It contains a functionality to check the status of the underlying mechanism for specified intervals.

NaradaBrokering offers stable storage to introduce state notion to the events [42]. Brokers do not keep track of the states. They are only responsible to assure the most efficient routing. Since brokers can possibly fail, NaradaBrokering introduces a stable storage for the recovering purpose. However, the guaranteed delivery scheme does not require every broker to have access to the stable storage.

The communications link may be insecure. Therefore, NaradaBrokering presents a strategy to secure the message transportation links. The scheme provides a framework to achieve end-to-end integrity while ensuring the authorized entities are the only ones that publish, subscribe and encrypt and decrypt the messages. The security framework is implemented in the context of centralized Key Management Center [43].

CHAPTER 3

DISTRIBUTED HANDLER ARCHITECTURE

Handlers are very necessary architectural components of Web Service Framework. They contribute to build a rich, modular, efficient architecture. However, the way of utilizing them is very essential to get full benefit from them. Distribution of the handlers among the individual physical and/or virtual machines provides many advantages and opens the doors through the immense computing resources. In the remainder of this chapter, we explain the general picture of Distributed Handler Architecture (DHArch) and provide the detail explanation about its modules.

3.1 General picture of Distributed Handler Architecture

Web Services are basically client/service interactions. They generally benefits from a middleware, called Web Service container. A container supports the interactions with the additional capabilities. Similarly, Distributed Handler Architecture (DHArch) is

a software system that provides functionalities to process handlers concurrently as well as sequentially within a distributed environment. It makes the handlers free from their boundaries and restrictions.

Web Service Framework uses various transportation mechanisms and protocols. The Hypertext Transfer Protocol (HTTP) is the one, mostly utilized. It is an application level generic stateless protocol for the distributed collaborative hypermedia information systems[12]. It provides a suitable communication environment among the organizations; by utilizing HTTP, web services can work through many common firewall security measures among the different organizations and platforms without changing any firewall policies. However, it also has some limitations especially because of the request/respond paradigm; the request has to be followed by a response. This results in an unnecessary network usage for some cases. It does not support asynchronous messaging very well. It necessitates an additional mechanism to achieve an asynchronous communication.

Consequently, a Message Oriented Middleware (MOM) has been chosen for the communication purpose [11]. MOMs are matured enough so that they guarantee the message delivery. A topic, context or query can be used to deliver a message. MOM offers asynchronous messaging between the computing nodes. It acts as a post office that carries the messages between the handlers and finally to the Web Service endpoint. It has reliable and secure communication means to carry out critical tasks. Moreover, it provides persistent storage capability; it can store the messages until they are delivered. Depending on the size of the message, thousands of messages can be queued to regulate the message flow.

In addition to the MOM usage, DHArch creates supplementary structures to carry out the message delivery and to keep the necessary information for the execution. Message is not the only entity that is necessary to be passed to handler; handlers may require more information for the execution. Hence, a relevant context is created to store that necessary information.

Moreover a handler orchestration is introduced to manage the distributed execution. DHArch may contain many computing nodes, which are distributed into the different environments. The orchestration among them is very important to perform the operations correctly. Therefore, An XML based-orchestration document is designed to describe the traversal of a message.

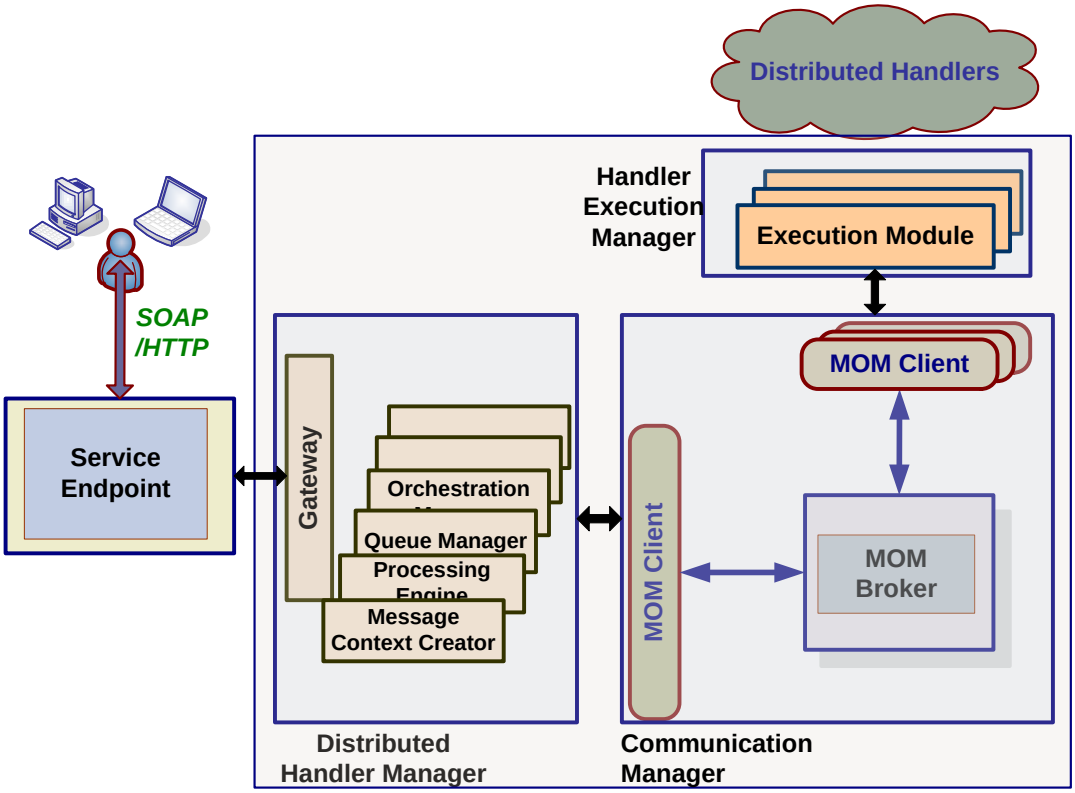


Figure 3-1: General Architecture of DHArch

Figure 3-1 depicts the general picture of DHArch. The handlers executing SOAP messages are distributed by using a Message Oriented Middleware. The messages travel between the distributed handlers by using the publish/subscribe mechanism offered by the middleware. It is perfectly capable of performing asynchronous messaging so that the request maker does not have to wait for the response. Instead, it continues its own tasks. The response is notified when it is ready.

While the architecture introduces new improvements, a very vital feature of Web Services is not ignored; the user knows only the service endpoint address and the service definition. DHArch is transparent to the user. It is not apparent to the user whether the handlers have been distributed.

DHArch contains many modules to manage message execution. Instead of having a very big chunk of hardly manageable implementation, DHArch employs several modules so that the implementation management became easier and more understandable. The next section explains the details of the modules.

3.2 Modules of Distributed Handler Architecture

DHArch modules can be placed under three umbrellas: Distributed Handler Manager (DHManager), Communication Manager (CManager) and Handler Execution Manager (HEManager).

3.2.1 Distributed Handler Manager (DHManager)

DHManager is an umbrella name for a group of modules that contributes to the message execution together. It is the hearth of DHArch. It basically accepts the messages, orchestrates the execution and returns the output to the place where the

message initially has been received. It contains several sub-modules: Gateway, Handler Orchestration Manager, Message Context Creator, Messaging Helper, Queue Manager and Message Processing Engine. We will explain these modules in the remainder of this section.

3.2.1.1 Gateway

Gateway is an interface between the native environment and DHArch. It is both entrance and exit point for the incoming and outgoing messages. Figure 3-2 portrays the functionality of Gateway. DHArch has a native environment independent architecture. It autonomously performs the given tasks. However, Gateway module is an exception and connects DHArch to the underlying environments by using the libraries and tools of those native environments.

An individual gateway is created for each interacting environment, which has its own execution structures. In order to cooperate, DHArch needs to employ a specific gateway. Hence, a new gateway component is necessarily constructed for every newly introduced SOAP processing environment. For example, Apache Axis and WSE require their unique gateways for the interaction.

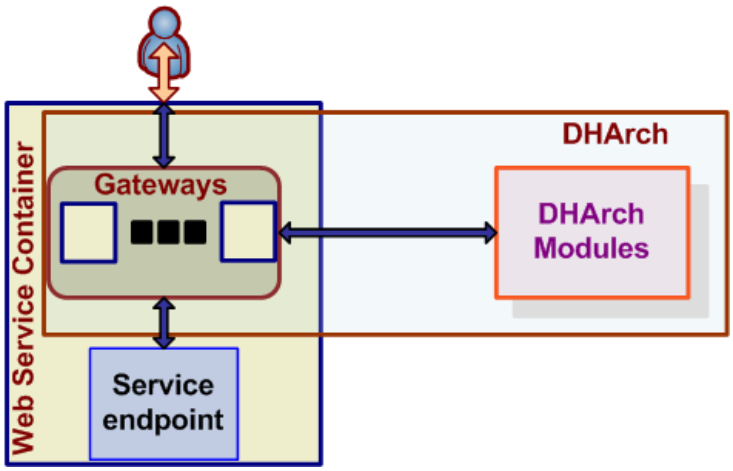


Figure 3-2 : DHArch Gateway

3.2.1.2 Handler Orchestration Manager

DHArch is a system enabling the execution of the handlers in a distributed fashion. By doing so, it provides a very fruitful atmosphere. This beneficial environment brings many advantages such as utilization of additional resources and concurrency. On the other hand, the distribution complicates the handler execution as a side effect. Handlers became unaware of each other when they are scattered around. Therefore, introducing an orchestration to manage the execution on the top becomes necessary.

Management of the distributed handler execution via an orchestration provides many advantages. It allows utilizing additional resources. Furthermore, handler usability increases by letting more than one service and client be able to use the same handler. Moreover, concurrent handler execution can be efficiently achieved via orchestration. The concurrency has extensively been investigated[44-46]. Bringing it into the Web Service handlers is worthy to struggle with the complexity of its management.

Even though many constraints are eliminated with the handler distribution, the requirements of the orchestration unfortunately generate challenges. Many investigations have already been accomplished in this area. We will explain our approach and challenges in CHAPTER 4.

3.2.1.3 Message Context Creator

A software system may introduce new features to facilitate the execution. DHArch contributes to the handler execution by utilizing a context, Distributed Handler Message Context (DHMContext), shown in Figure 3-3. This context wraps every request arriving to DHArch. It also keeps supplementary information for the message execution.

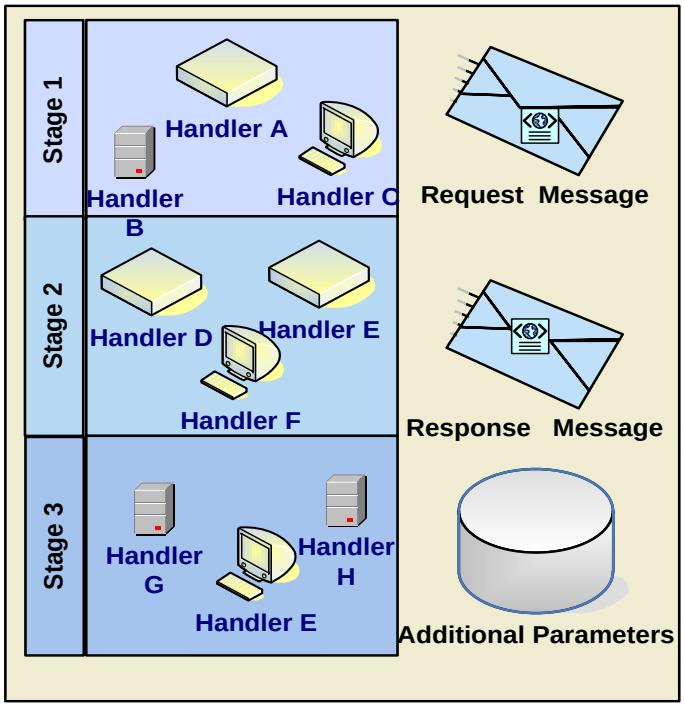


Figure 3-3 : Distributed Handler Message Context

Moreover, the handler orchestration configuration is kept in this context to provide a convenient way for the handler executions. With this structure, every message can have its unique handler orchestration configuration, which employs handlers and stages. Stages are the places where the parallel execution happens. Each message has at least one stage in its execution chain. Similarly, each stage must contain at least one handler. Default stages, their corresponding handlers and the necessary parameters for the execution are initiated by using the orchestration document, explained in CHAPTER 4.

Moreover, DHMContext contains additional parameters to facilitate the execution. The current stage number, the number of handlers, the number of stages, start and end times of a handler execution, boolean variables for the handler or stage executions and so on are necessary to process the handlers correctly.

The current stage number keeps track of the executing stage at a moment for a message. An execution may contain many stages and they need to be processed in an order. By contrast, handlers within a stage do not have to be executed in an order. They are executed in a parallel manner. Therefore, keeping track of a sequence number for the handlers is not necessary. Instead, the number of handlers employed in a stage is required to finalize its execution.

Start, end and elapsed times of a handler execution are gathered for the monitoring purpose. These timing variables contribute to build a reliability mechanism for the stateless handler executions too. An execution is repeated for a stateless handler, when it cannot be completed within a reasonable duration. If it isn't successfully achieved in several trials, it is concluded that either the distribute handler is down or the network connection is broken.

The orchestration structure in DHMContext is initialized by using an XML document. It can be modified during the execution. This runtime update allows a dynamic handler and stage execution. In order to modify the execution, a modifier needs to be employed. It is a specialized handler that looks into the message and finds out which handlers are necessary for the execution. In short, a message is able to decide its execution flow by utilizing a modifier.

An execution of a stage cannot be completed unless the constituent handlers finalize their tasks. There are two exceptions for this mandatory situation: *one-way* request and having false value for *mustPerform*. Some handlers may not need to send a response or acknowledgment back such as logging handlers. It is appropriate to apply fire and forget paradigm for this kind of handlers. Additionally, some handlers may not be so

essential for the execution that skipping its execution is acceptable. Hence, even if the handler fails completing its task, the execution can continue when mustPerform is false.

3.2.1.4 Queue Manager

Sometimes, a Web Service receives too many requests in a short duration so that the service requests cannot be answered. For this reason, queues are introduced to regulate the message flow. It is similar to having a waiting room in a doctor office. When a patient arrives, s/he is asked to fill the necessary information and to be seated in the room until the doctor becomes available. Similarly, DHArch registers the information of a message into DHMContext and makes the message wait to be called.

Queue Manager manages the acceptance of the messages. It employs three queues to prepare a message for the execution. The first queue, Container Message Context Queue (CMCQueue), stores the interacting Web Service container contexts. The queue allows storing the different type of objects. For example, *MessageContext* is the context object of Apache Axis container. Since other Web Service containers employ their unique contexts, the queue is able to provide storage for them too. By storing a context, the required information is saved so that an interacting Web Service container with DHArch is facilitated to continue its processing.

Every context is registered with a 128-bit unique key, created by a UUID generator. It is used to identify the corresponding message in DHArch. The uniqueness keeps the message execution intact; the message execution cannot possibly interfere with another. Therefore, there is no chance of blending the execution of an individual message.

In addition to CMCQueue, the system maintains two more queues: Incoming Message Queue (IMQueue) and Message Processing Queue (MPQueue). These queues store Distributed Handler Message Context (DHMContext), which is created by Message Context Creator module.

For each arriving message, IMQueue stores a DHMContext object. It is a First In First Out (FIFO) queue[47]; while a new message is inserted to the tail of the queue, the execution engine pulls a message out from the head. FIFO queue is chosen because it is a fair data structure; it equally treats the messages. In other words, the first arriving request message has the biggest priority. However, the queue can be easily adapted to the other schemes. A priority based queuing scheme can be used to improve the system performance or to provide new capabilities.

The third queue, MPQueue, is the place where the message processing happens. The number of the messages in the queue is limited to optimize it. The queue size is very small compared with previous queues. It provides a regulated pipelining capability for the message executions; the messages are executed concurrently and the regulation prevents hurting the system performance with overwhelmingly pipelined message execution.

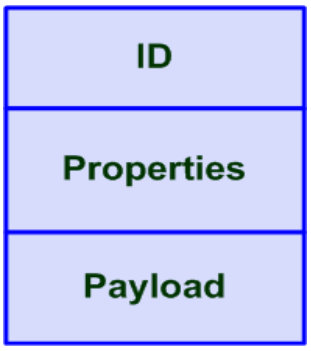
3.2.1.5 Messaging Helper

Messaging is a very significant capability for a system to decouple the computing nodes. Sending and receiving the tasks among the interacting nodes via messaging contributes to the interoperability. Message Oriented Middleware (MOM) offers an excellent environment for this purpose. Although the messages can be sent in different formats, a specific format, DHArch Messaging Format (DMFormat), is used to facilitate the remote handler executions. Figure 3-4 depicts that format, which conveys the

necessary information to the distributed handlers. It basically contains three main parts, unique ID, properties and the payload.

DHArch may host many messages being executed in a moment. Hence, an identifier is a necessity to achieve the correct executions; a UUID [48] generated identifier is assigned to every message. The execution mechanism recognizes the messages from their IDs. The generator assures that there won't be the same identifier in the system. Thus, the system gives enough guarantees that the message executions are not blended.

The second important part of DMFormat is the properties section. This part conveys the required information for the computing nodes: Handler Execution Manager (HEManager) and Distributed Handler Manager (DHManager). The transferred information can be specific to a handler as well as generic for all handlers. For example, a property can be defined for a *one-way* handler so that it does not have to send an unnecessary message back. Many other properties can be added to contribute handler execution. We utilize any type feature to support this extensibility.



```
<context>
  <id>
      4099d6dc-0b0e-4aaa-95ff-2e758722a959
  </id>
  <properties>
   <oneway>false</oneway>
   <mustPerform>true</mustPerform>
  </properties>
  <payload>
          .....
  </payload>
</context>
```

Figure 3-4 : DHArch Messaging Format

Extensibility of the properties supports to convey supplementary information. Properties can also facilitate the communication among the distributed handlers directly. Since we utilize a publish/subscribe mechanism of NaradaBrokering, the next handler address, defined in the orchestration, can be passed in a property. DHArch mainly supports the centralized approach; it sends a message to a handler and then expects to receive the response back before starting the new handler execution. However, if handler executions are sequential and the next handler in the chain is known, the request can be forwarded to the next handler from the executing one instead of using DHManager as an intermediary node.

The third part of DMFormat is payload. The payload contains the original message. There isn't a restriction for the payload format; any kind of message format can be embedded to the payload. The only restriction is that it should be comprehensible by the targeted handler.

DMFormat is utilized to send and receive the messages. During the response, a DMFormat is fabricated again with the same unique ID for the response. The properties

may be modified. The payload is not a request message anymore; it carries the response at this time.

3.2.1.6 Message Processing Engine

Message Processing Engine (MPEngine) is the maestro of the system; it orchestrates the executions. It employs three threads to accomplish the following important tasks: selecting candidate messages, sending messages to the distributed handlers and receiving the responses. In short, the name is not a coincidence; the main activities of the handler processing management are carried on by this module.

Message Selector Thread (MSThread) selects a candidate message from Incoming Message Queue (IMQueue) to start the execution. The candidate message is placed into Message Processing Queue (MPQueue). Two events trigger this message selection. The first event is to have fewer messages than the optimum number of messages in MPQueue. The second triggering event is the new message arrival. This is important in the situation that IMQueue becomes empty while the MPQueue contains less than the optimum number of messages.

When a message context is moved to the MPQueue, it means that the message is ready to be executed. There exist two threads operating over this queue. The first one is Message Processing Thread (MPThread). As we discussed earlier, the queue consists of DHMContext objects. The contexts are very critical to invoke the handler executions in the right order. MPThread looks to the queue from the beginning to the end. It takes the contexts and extracts the required information for the executions. The thread checks the flow structure that tells the engine where to send the messages. When everything is ready, the transportation of the messages to their destinations is initiated.

The message transportation is handled by Communication Manager (CManager). MPEngine passes the messages to CManager according to the orchestration structure. A message can be sent to the multiple handlers at once in order to have parallel execution as well as it can be passed to the handlers sequentially.

When a response message is received from the CManager, the third thread of the MPEngine, Message Receiver Thread (MRThread) is activated. It initially checks the message ID. If there is a match, the corresponding context in the MPQueue is searched and finally the context is updated with the incoming response. When the handler executions are completed, in other words every handler finishes its task, the context is removed from the queue by MRThread. At the same time, the corresponding container context is extracted from CMCQueue by using the same message ID. The final task of the thread is to combine the executed message with the Web Service container context. At this point, MRThread completes its task and returns the container context to DHArch Gateway.

3.2.2 Communication Manager

Communication Manager (CManager) transports the messages between the computing nodes. It utilizes pub/sub paradigm and comprises of subscribers and publishers to send and receive messages. A Message Oriented Middleware (MOM) is employed for the transportation. MOM is mature enough to achieve very critical tasks. We use NaradaBrokering for this purpose [49]. It provides many key advantages for the internal messaging.

The first advantage is asynchronous messaging. There exist many researches in this area [50-53]. It supports to decouple system components. The sender and receiver do

not oblige to be presented together during the execution. While the requester asking a service, the provider can be in the situation of performing another job. This is also called as non-blocking IO [54]. In many places, these two notions are used interchangeably. Asynchronous messaging utilizes none-blocking IO between the computing nodes. The requester does not wait for the result; it is notified when the output is ready. This eliminates the idle waiting.

The second advantage is the usage of NaradaBrokering to regulate the message flow. Flow control has been widely investigated [55-58]. There might be durations that the system is overloaded with the incoming messages; during peak times, the message rate may be so high that the receiver cannot handle all the messages. This is similar to building a water dam that is used to irrigate the agricultural areas. During peak season, the water current may be so high that can flood the area and cause damages. Water is necessary for the irrigation when the water is scarce. So having a dam serves for two purposes, two birds with one stone: preventing flooding and providing abundant water when it is needed. Similarly, NaradaBrokering acts as an irrigation system that has a dam to control the flow. It can buffer as many messages to overcome the flow of peak times. It releases these messages so gradually that the receivers are able to handle the messages. We conducted an experiment for this purposes, NaradaBrokering can keep up to 10000 messages in the pipe depending to their sizes. However, this is very promising for our system so that we cannot possibly reach even this limit.

The third advantage is efficiency. One of the main concerns in publish/subscribe systems is the performance because of the usage of additional player between two peers.

However the benchmarking results show us that NaradaBrokering is so efficient that it meets the demands of ours. The detailed performance results prove our claim[59].

The fourth advantage is to have a guaranteed message delivery mechanism by utilizing NaradaBrokering [60]. Reliability is important to make sure that a message is delivered to the destination. There exist many researches in this area[61, 62]. Web Service community has recently introduced specifications for the reliable communication [19, 22]. In the distributed handler mechanism, messages are so essential for the execution that the system must have a structure that guarantees the deliveries to the destinations. NaradaBrokering provides a robust delivery mechanism by storing messages in a database so that the peers can get them later even if a failure occurs. However, the utilization of the robust node causes an additional cost. Even though we reserve the right to use this capability, the utilization of the normal delivery mechanism is selected for the delivery because it offers enough reliability. Moreover, an additional reliability mechanism is built on the top of this transportation level reliability. We will explain this in CHAPTER 5.

NaradaBrokering scales very well because of tree structure broker network capability. Many brokers can link together to build a tree. There might be a situation that one broker can saturate that the handlers cannot be supported efficiently. This limitation can be gotten rid of with the introduction of a new broker. Even the performance can be improved with this approach; handlers can be grouped around a broker that shortens the communication distance. This structure helps to build an efficient transportation environment for the distant handler deployments.

The last but not least, NaradaBrokering employs publish/subscribe mechanism for the message delivery [63]. In a typical publish/subscribe system, there is a topic that publisher sends to and subscriber receives from. NaradaBrokering offers more sophisticated mechanisms. In addition to topic base matching algorithm, NaradaBrokering allows utilizing content base matching by leveraging XPath query for XML base documents [39].

Consequently, Communication Manager (CManager), depicted in Figure 3-5, provides an efficient transportation media. It uses an efficient publish/subscribe mechanism. Publish/subscribe paradigm is benefited as follows; every computing node has its own topic. In other words, every computing node is uniquely addressable. The messages are sent to those addresses in an order. The topics are mapped with the handlers before communication is started. In a parallel execution, we may assign one topic to the handlers that are concurrently executed. While this reduces the number of addresses in the system, it also prevents modifying the execution flow on the fly. Therefore, we stick the paradigm of a single topic usage for each handler whether it is a parallel or sequential execution. DHArch is able to possess the replicas of a handler. The URI base topic structure is the best for this kind of execution. The replicas' topics consist of two parts; the first part is exact match to show that they are replica. The second part is individual so that they can be differentiated from each other.

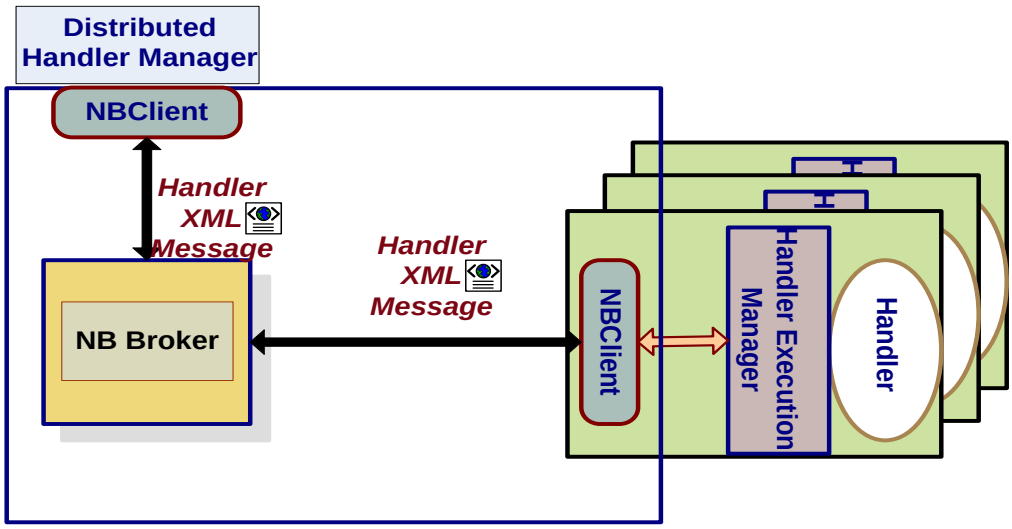


Figure 3-5 : DHArch Communication Manager

CManager utilizes a NaradaBrokering client containing a subscriber and a publisher for the distributed handlers. The manager assigns an instance from this client to each handler as well as one to Distributed Handler Manager. Clients are the entry/exit points for the distributed handlers; the messages are passed through these clients. Similarly, the executed messages are sent back to the CManager with these clients.

CManager utilizes a special message format created for one sole purpose: transferring the messages efficiently. Even though the message size increases because of the side effect of the usage of XML based format, it contributes to the execution in many ways when the message reaches the destination. This format has been explained in detail in section 3.2.1.5.

3.2.3 Handler Executing Manager

Distributed handlers are the applications executing the messages in the remote places. Without having a supportive environment, they cannot perform their tasks.

Handler Execution Manager (HEManager) is considered to build this necessary environment. Each distributed handler is hosted by a HEManager. It supports the execution in several ways, stretching out from negotiating with CManager for the communication to creating the necessary structures.

HEManager is the component that accepts the messages and forwards them to the final destination, a distributed handler. There are as many HEManagers as the number of distributed handlers. Both incoming and outgoing messages travel with DMFormat, explained previously. When a message arrives to a node, the essential information is extracted and necessary structures are constructed for the handler execution. The structures are built around the unique ID, the name of a message in DHArch. HEManager facilitates the ID to prevent confusing on the execution. One additional advantage of the unique ID is that the manager knows to whom it will reply.

The execution greatly gets assistance from the properties, an element of DMFormat. This element carries the necessary data for the handlers. However, they are not the executable data. Instead, they convey the information for the executers. Supplementary properties can be transferred in this element. However, the handler should be aware of how to deal with these custom properties.

HEManager leverages the common interfaces to standardize the handler implementation. A handler can be easily implanted to DHArch as far as it implements these interfaces. Moreover, HEManager support some well known handler interfaces such as Apache Axis handler interface. Consequently, Apache Axis handlers can be plugged into HEManager seamlessly.

A typical handler expects to receive a SOAP message as an input. However, it is not limited with SOAP format. The input can be any XML element; it can even be a partial SOAP message. Our intention is to reduce the transportation cost where the necessary data for the handler execution is a small friction of the whole message. If a handler is far away from the Web Service, the system will be more efficient if the size of the transmitted message can be made smaller.

The output is returned to HEManager when the distributed handler completes the execution. The manager wraps the message with DMFormat by using the same unique ID. Otherwise, the response cannot reach the right destination. The orchestration is kept hidden from HEManager with the intention of keeping its execution simple. It only knows how to create the environment for the distributed handler and where to send the response coming out of the execution.

3.3 Summary

In this chapter, we explained DHArch general concept and its modules. DHArch has a modular architecture which improves the maintainability and simplicity. The modules can be classified into three sub-groups. The first group contains the modules located in a place which Web Service endpoint resides. These modules supervise the orchestration and monitor the correctness of the execution. The second group performs the transportation of the tasks between the computing nodes. Finally, the last one is to create an environment for the distributed handlers.

CHAPTER 4

DISTRIBUTED HANDLER ORCHESTRATION

Web Service is defined by W3C as a software system that provides a standard means of interoperating the different software applications, running in a variety of platforms[1]. It utilizes an interface, WSDL, to interact with the clients. There are two important nodes for a Web Service interaction: provider and requester. Web Service architecture employs a SOAP processing engines and transport helpers to contribute the interaction. These functionalities are generally provided by a middleware called Web Service container. A container essentially hides the complexity of the SOAP processing and the details of message transportation.

Web Service architecture employs additional functionalities to utilize the extensibility feature of SOAP. The functionalities provide new additive capabilities to Web Services. Depending on the Service container, these capabilities called handlers or

filters. Generally, a Web Service container provides a handler processing pipeline so that many handlers can contribute to Web Service interactions.

Handlers are able to become autonomous processing nodes. They can be detached from the Web Service endpoint with the intention of creating more powerful, efficient, scalable and modular Web Service environments. Web Service architecture is suitable for this separation so that the correctness of the execution is not harmed.

When handlers are separated from a Web Service endpoint successfully, they become individual applications running without knowing each other. There are many reasons to separate a handler. We may need to benefit from more resources such as processor power, memory, and storage space. We may have a powerful architecture by offering a more modular and scalable structure. We may increase usability. Finally, we may successfully introduce concurrency to the handler execution. However, all these advantages do not come for free. The detached handlers are needed to be orchestrated so that they can achieve the execution, which was successfully happening before the separation. At this moment, the notion of handler orchestration comes to light. We will discuss the handler orchestration in the remainder of this chapter.

4.1 Workflow systems

Workflow languages and systems provide means of accomplishing some or all of the tasks in a distributed environment. Hence, a flow mechanism benefits several supportive structures. The first one is to represent the dependencies of the services. These can be either temporal or data driven dependencies. The second structure is to provide the necessary constructs to control the execution such as conditional branching or looping. Finally, a flow mechanism requires the management of the scheduling and the execution.

Workflow Management Coalition (WfMC) worked hard to come up with an agreement to standardize the workflow efforts. WfMC mission is to support workflow systems and create the standards. After spending many efforts, a workflow reference model has been emerged. WfMC has defined and explained the major components and interfaces. According to this model, the core of a workflow mechanism is workflow enactment service. This service may contain several engines to control and execute a workflow. Every engine can operate on a selected part of the workflow. To specify and analyze the workflow, the reference model requires a process definition tool, in other words, a routing definition. The definition describes which tasks need to be executed and in what order. A Workflow client application may be leveraged to interact with the system to submit tasks. Workflow mechanism also utilizes monitoring and controlling tools. These tools facilitate to find out the bottlenecks and to register the events for later usage[64].

Additionally, WfMC defines a common set of terms for Workflow developers, researchers and vendors. The following definitions described by WfMC are important to understand the concept of the routing constructs[65]:

Parallel execution: *"A segment of a process instance under enactment by a workflow management system, where two or more activity instances are executing in parallel within the workflow, giving rise to multiple threads of control."*

Sequential execution: *"A segment of a process instance under enactment by a workflow management system, in which several activities are executed in sequence under a single thread of execution."*

Iteration*: "A workflow activity cycle involving the repetitive execution of one (or more) Workflow activity(s) until a condition is met."*

Conditional execution*: "A point within the workflow where a single thread of control makes a decision upon which branch to take when encountered with multiple alternative workflow branches."*

Many efforts have been spent to obtain a system providing a solution to manage tasks and data in the distributed environments. The academic community joined these efforts to orchestrate the complex tasks within the distributed environment. GriPhyn[66] provides a good computational environment for the particle physics. SEEK[67] has a solution to orchestrate the tasks for ecology. Taverna[68] offers a flow mechanism for life science. Not only did the academic community provide a solution but there also exist propriety software for the distributed task management. Inconcert[69] , Websphere MQ Workflow[70], Lotus Workflow[71] are the examples of the systems in the market.

Moreover, Grid community has an interest in this area because of their focuses on secure and collaborative resource sharing across geographically distributed institutions. GridFlow[72] offers an agent-based architecture to schedule the Grid tasks dynamically. GridAnt[73] is a workflow mechanism motivated to develop a simple, extensible, platform independent, and client controllable workflow mechanism. Additionally, several new specifications have been presented such as Business Process Language for Web Services (BPEL4WS) [74], Grid Service Flow Language (GSFL) [75] and Web Services Choreography Interface (WSCI)[76].

There are several ways to provide a workflow mechanism for job coordination. Workflow can be defined by software components. In other words, a workflow

mechanism can be hardwired. However, this may cause a trouble when the execution sequence needs to be altered. Instead, the workflow should be defined above the underlying software mechanisms with an appropriate semantics. There are three main approaches for this purpose. The workflow may be based on a scripting language such as GridAnt, JPython[77] and XCAT[78]. It may also utilize graphs as it is in Condor DAGman[79] and Symphony[80]. Finally, there are workflow applications that utilizes both approaches such as XLANG [81], WSFL[82].

Many workflow mechanisms leverage Directed Acyclic Graph (DAG) such as UNICORE[83], Condor and Cactus. It is defined as *"a directed graph with no directed cycles that is for any vertex v, there is no nonempty directed path starting ending on v."* The advantage of DAG is its simplicity. Therefore, it is preferred in many applications. However, DAG is acyclic. Hence, defining a loop is impossible. It also only describes the behavior. A system state cannot be monitored.

Therefore, many workflow mechanisms chose Petri-net based model for the orchestration. It builds a graphical definition of a workflow by using few simple graphical elements. This graphical interface is converted to a comprehensible output for the workflow engine. This output may be any kind of document that the workflow engine understands. For example, Grid Job Builder creates a GJobDL document that defines the Grid Job [84].

Petri net [85] is defined as "*a directed graph with two kinds of works , interpreted as places and transitions, such that no arc connects two nodes of the same kind*" [86]. It is able describe the flow activities of a complex system. Synchronization, parallelism, sequential processing and conflicts can be effectively modeled. It basically contains the

places and transitions connected together by arcs. A place is represented by a circle while a transition is symbolized by a rectangle. In spite of its simplicity graphic-wise, it perfectly represents a flow mechanism for complex systems.

Although Petri net is a graphical representation that can be used in practice, it is also a precise mathematical model. Here is the formal mathematical definition [64].

A Petri net is a triple (P, T, F) where

- P is a set of places
- T is a finite set of transitions ($P \cap T = \emptyset$)
- $F \subseteq (PxT) \cup (TxP)$ is a set of arcs

Petri net models the behavioral aspects of a system. Therefore one of the most important advantages of using Petri net is that the behavioral aspects of a distributed system can be easily described. The components that are separated locally can be illustrated without difficulty.

There are several extensions of Petri net model. Although the extensions provide an additional power for the modeling, the compatibility problem may occur because of missing universally accepted objects in the model. Colored Petri net prioritized Petri net, timed Petri net are among its extended versions.

4.2 Orchestration and its XML schema

Orchestration is the key feature of building an efficient distributed handler execution. Hoping that handlers hosted by different computers can intelligently execute messages without an orchestration is not reasonable. The distributed execution needs to be facilitated so that the sequence of the execution can be understood by the handlers.

Additionally, with the contribution of the orchestration, the handlers can be liberated from their limited surroundings and they can benefit from new features and resources.

Fortunately, we utilize SOAP messages in Web Services. It conveys metadata with data together. This feature minimizes the reference issue. Distributed computing requires a good referencing model for data and computing nodes. The referencing is one of the most challenging problems. The general solution of referencing a data object is the usage of pointers. However, there exist limitations in this solution. The referenced object may not be created again so that the pointer loses the object when the application needs to be restarted. Many solutions have been introduced for this problem. DISCWorld is an example and provides high level middleware to access to data and resources. It utilizes canonical names for the objects. When user makes a request, it is analyzed by a local IDSCWorld daemon. The daemon invokes a placement algorithm to assign the services to the processing nodes [87]. In Web Services, luckily, SOAP messaging is utilized. Therefore, data referencing becomes issue no more. The only entity that needs to be referenced is the handler so that the messages can be passed properly. Handler referencing is explained in section 3.2.2.

There exist many workflow systems that utilize markup languages. One of them is The Petri Net Markup Language (PNML) [88]. It makes Petri net model transferable so that users take advantage of newly developed facilities such as simulation, analysis and implementation tools. The main design principles of PNML are flexibility and compatibility. The idea is that it should not limit the features of any kinds of Petri net and be able to represent every Petri net model with its extensibility features. It also provides an effective compatibility with the well defined labels [89].

Additionally, several other projects benefits from the markup languages. eXchangeable Routing Language (XRL) uses XML base documents for the workflow management [90]. The language consists of basic routing structures that can be utilized to design more complex routing schemes.

Markup languages clearly provide many opportunities. DHArch handler orchestration mechanism also utilizes an XML based document to describe the sequence and the resources. XML carries semantic as well as syntax. This feature allows the document to be interpreted by other systems; additional tools and software can be utilized. In order to define the orchestration document, an XML schema is created. XML schemas describe the structure, content and semantics of XML documents [91]. They define the shared vocabularies of instances of XML documents. Now, we will explain the handler orchestration document schema:

Table 4-1: Simple elements in Orchestration Schema

```
<!--Element Definitions-->
<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="oneway" type="xs:boolean"/>
<xs:element name="mustPerform" type="xs:boolean"/>
<xs:element name="condition" type="xs:anyType"/>
<xs:element name="numberOfHandler" type="xs:short"/>
<xs:element name="numberOfLooping" type="xs:short"/>
```

DHArch handler orchestration schema contains several simple and complex elements to define the flow sequence, shown in Table 4-1. Simple elements contribute to build complex schema elements. Name, address, oneway and mustPerform are the elements to define a handler. Condition, numberOfLooping and numberOfHandler support to fabricate the execution constructs.

A time entity is necessary to monitor handler states. Therefore, a complex type is built for it, shown in Table 4-2. Several time-related variables are required to construct a handler. Start, end and execution times can be necessary to watch a handler execution. The instance of time element includes the definition and the value. A handler may use many time instances as well as it may not include any.

Table 4-2 : Complex time element

```
<xs:complexType name="timeType">
        <xs:sequence>
                <xs:element name="definition" type="xs:string"/>
                <xs:element name="timeElement" type="xs:long"/>
        </xs:sequence>
</xs:complexType>
```

Handler is the most important entity of the orchestration schema. In other words, it is the keystone of an orchestration. Table 4-3 defines a handler. It is composed of several elements. The name is an identifier to increase readability of the document by the user. A handler must have a unique address so that a message can be delivered to. We keep tract of the time related data for a handler to collect statistic data and to assure the message delivery. We also have additional information to support handler execution.

Table 4-3 : Handler Definition

```
<!--Defines Handler-->
<xs:complexType name="handlerType">
        <xs:sequence>
                <xs:element ref="name"/>
                <xs:element ref="address"/>
                <xs:element ref="mustPerform"/>
                <xs:element ref="oneway"/>
                <xs:element name="time" type="timeType" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
```

The schema defines four basic constructs, shown in Table 4-4. The complex execution structures are composed from these basic constructs. They are sequential, parallel, looping and conditional. There may be only one of them as well as many of them in an orchestration. Each construct has a position parameter to identify its order in the execution. The constructs are sequentially ordered; they are processed in the order defined by the position element.

Table 4-4 : The execution constructs

```
<xs:element name="executionConstruct">
        <xs:complexType>
            <xs:choice>
                <xs:element ref="sequential"/>
                <xs:element ref="parallel"/>
                <xs:element ref="looping"/>
                <xs:element ref="conditional"/>
            </xs:choice>
            <xs:attribute name="position" type="xs:short" use="required"/>
        </xs:complexType>
</xs:element>
```

Many execution constructs get together to build an execution sequence. In the next section, we will explain the basic constructs.

4.3 Execution constructs

The materials in the universe are composed from the elements defined in the periodic table although their numbers are limited. A written document comprises only letters that are defined in an alphabet. A software language has a small set of basic types to build up a complex syntax. A processor contains the small set of instructions to execute the complex commands. The same concept can be applied to the handler orchestration.

The common feature of the chemical elements, an alphabet, the basic types of a language and a processor instruction set is being well defined. In order to construct more complex structures, the basic building blocks must be well defined. Hence, the four basic constructs of DHArch orchestration mechanism is well defined and has the power to address complex execution patterns.

Moreover, the schema is compatible with Petri net model so that the orchestration benefits from a workflow and mathematical model. We will provide the graphical representations of the constructs in the Petri net model to show the compatibility. The intention is not to make a research about the applicability of the Petri net model to the distributed handler execution in this dissertation, though. We want to show the possibility of utilizing the model. Now, we will explain the basic execution constructs:

Table 4-5 : The sequential secution construct

```
<xs:element name="sequential">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="handler" maxOccurs="unbounded"/>
                        <xs:element ref="numberOfHandler"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Table 4-5 explains the definition of the sequential execution. A construct must contain at least one handler. The order of the execution depends on the position of the handlers in the construct. Figure 4-1 depicts Petri net model representation of the sequential execution construct.

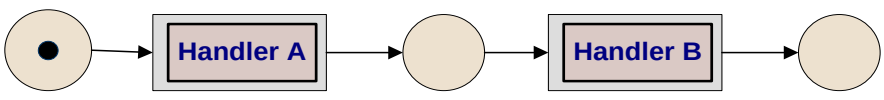


Figure 4-1 : Sequential Execution Petri net representation

The parallel execution, shown Table 4-6, is more complicated than the sequential one. There exist several types of parallel execution. A synchronous execution forces the engine to finish every handler execution before starting the next constructs. On the other hand, in an asynchronous execution, the next constructs may start their executions, before the completion of the handlers in a contract.

Table 4-6 : The parallel execution construct

```
<xs:element name="parallel">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="handler" maxOccurs="unbounded"/>
                        <xs:element ref="numberOfHandler"/>
                        <xs:element ref="typeOfParallelExecution"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

In order to have parallel execution, there must be at least two handlers in a construct. The upper bound is not set. The parallel execution Petri net representation is depicted in Figure 4-2.

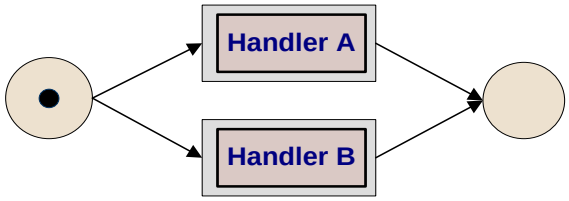


Figure 4-2 : Parallel execution Petri net representation

Some handlers may need to be processed repeatedly. Instead of having multiple appearance of a handler, the number of looping can be provided to have a neat document structure. Table 4-7 shows the schema representation for the looping construct. The

quantity of the handlers in a loop is basically one. However, a set of handlers may be processed together many times. In other words, many handlers can also be in a loop.

Table 4-7 : The looping execution construct

```
<xs:element name="looping">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="handler"/>
                        <xs:element ref="numberOfLooping"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

The Petri net representation of the looping is depicted Figure 4-3. The execution starts from the place containing the token. It ends in the place after the *handler A* transition. This process repeats itself.

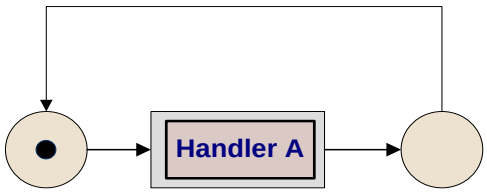


Figure 4-3 : Loop execution Petri net representation

An execution may require selecting a handler from a set of handlers according to a condition. Condition describes which handler is going to be chosen for the execution. Any type XML element is used to represent conditional construct. Table 4-8 illustrates the conditional handler execution construct.

Table 4-8 : The conditional execution construct

```
<xs:element name="conditional">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="handler" maxOccurs="unbounded"/>
                        <xs:element ref="condition"/>
                </xs:sequence>
```

```
        </xs:complexType>
</xs:element>
```

Figure 4-4 depicts Petri net representation of the conditional handler execution. An additional place is used to symbolize the condition. When the execution reaches the condition, only the transition of the chosen handler is processed.

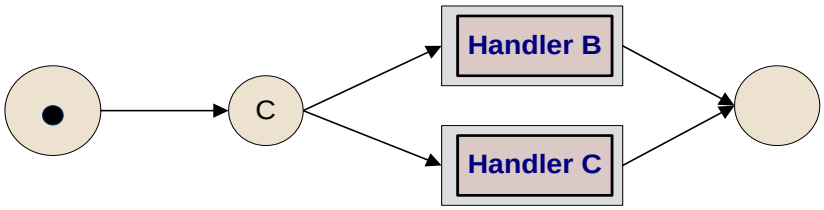


Figure 4-4 : Conditional execution Petri net representation

4.4 A handler execution scenario utilizing basic constructs

We create a handler orchestration instance, depicted in Figure 4-5 , to elaborate how to construct an execution of distributed handlers. We intentionally put a single occurrence from each basic constructs. The first construct consists of three handlers running sequentially. The second construct contains four handlers processed concurrently. Each handler starts their executions at the same time while they may complete them in different moments. Depending on the type of the parallel execution, the engine may have to wait the completion of each handler in the construct. The third construct is a looping which the instances of a handler are executed sequentially. Finally, conditional execution is employed to select a handler among a group of handlers. There might be several conditions for a construct.

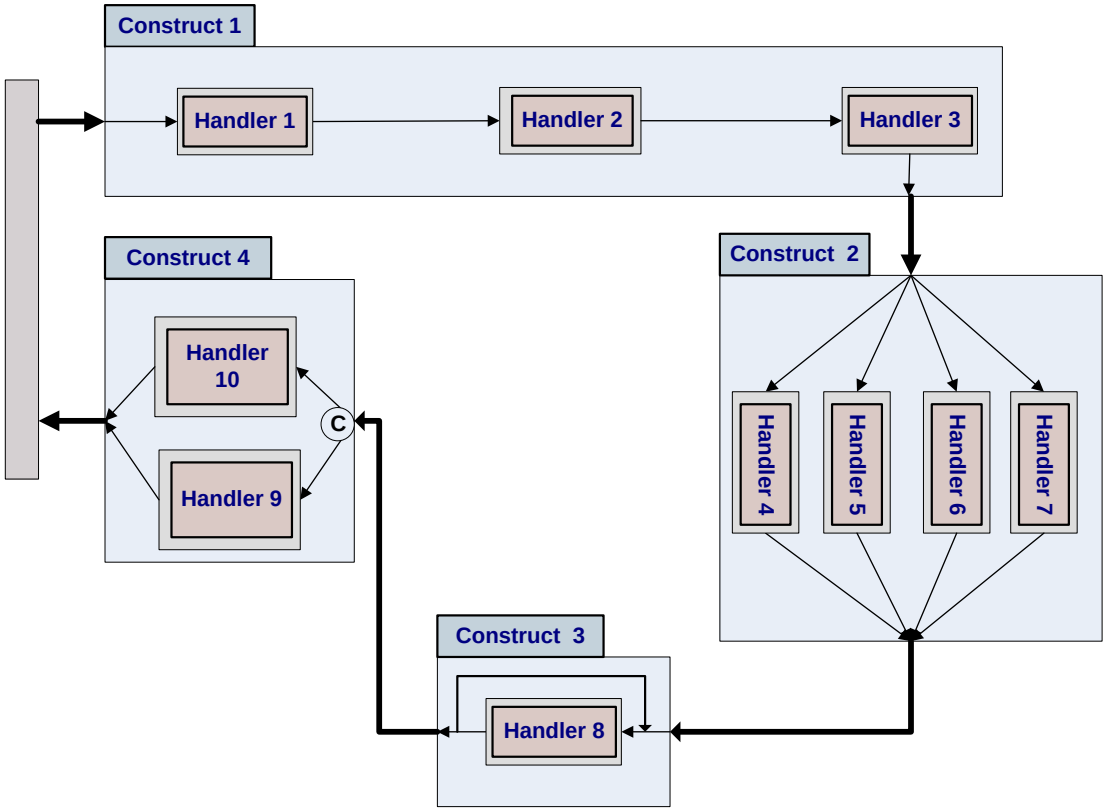


Figure 4-5 : A sample of a handler orchestration

Each construct has its own XML representation; we will explain the details in the remainder of this section. Table 4-9 contains the XML serialization of three handlers running sequentially. A handler must consist of its unique address. The order of the execution is defined by the appearance of the addresses. The position attribute defines the execution place of the construct among the remaining constructs.

Table 4-9 : A sequential execution serialization

```
<executionConstruct position="1">
        <sequential>
            <handler>
                <name>handler 1</name>
                <address>/dharch/handler1</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
            </handler>
```

```
            <handler>
                <name>handler 2</name>
                <address>/dharch/handler2</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
             </handler>
             <handler>
                <name>handler 3</name>
                <address>/dharch/handler3</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
             </handler>
              <numberOfHandler>3</numberOfHandler>
        </sequential>
</executionConstruct>
```

The snippet in Table 4-10 describes the parallel execution of the four handlers. Each handler has a unique address and supplementary information for the execution. The order of the handler execution is not crucial because the executions start at the same time. There are two types of parallel execution: *synch* and *asynch*. In *sync* execution, every handler should finish their executions to start the execution of the next construct. However, this is not obligatory in the *asynch* type parallel execution.

Table 4-10 : A parallel execution serialization

```
<executionConstruct position="2">
        <parallel>
            <handler>
                <name>handler 4</name>
                <address>/dharch/handler4</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
            </handler>
            <handler>
                <name>handler 5</name>
                <address>/dharch/handler5</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
             </handler>
             <handler>
                <name>handler 6</name>
```

```
                <address>/dharch/handler6</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
            </handler>
            <handler>
                <name>handler 7</name>
                <address>/dharch/handler7</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
            </handler>
            <numberOfHandler>4</numberOfHandler>
            <typeOfParallelExecution>synch</typeOfParallelExecution>
        </parallel>
</executionConstruct>
```

Table 4-11 shows a looping construct. The number of loops describes how many instance of a handler is processed sequentially. Although a single handler looping is the main target, a group of handlers can utilize the looping too. In a group looping, there is more than one handler processed repeatedly.

Table 4-11 : A looping execution serialization

```
<executionConstruct position="3">
        <looping>
            <handler>
                <name>handler 8</name>
                <address>/dharch/handler8</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
            </handler>
                <numberOfLooping>2</numberOfLooping>
        </looping>
</executionConstruct>
```

A handler orchestration can facilitate a conditional execution when a decision is necessary. Depending on the given condition, execution path is decided. The construct, depicted in Table 4-12, portrays a conditional execution of two handlers. A *condition* may describe both the condition and the action. For example, the snippet describes that

handler 9 needs to be executed when the SOAP message contains *wsLog* element. A conditional construct can have more than one condition element.

Table 4-12 : A conditional execution serialization

```
<executionConstruct position="4">
        <conditional>
            <handler>
                <name>handler 7</name>
                <address>/dharch/handler7</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
             </handler>
             <handler>
                <name>handler 7</name>
                <address>/dharch/handler7</address>
                <mustPerform>true</mustPerform>
                <oneway>true</oneway>
             </handler>
             <condition>
                  <isElementExist elementName="wsLog">handler 9</isElementExist>
            </condition>
        </conditional>
</executionConstruct>
```

4.5 Interpretation of an orchestration document

DHArch engine interprets an XML base handler orchestration document and creates its internal execution structure to carry out the handler processing. In other words, the constructs of an orchestration document are mapped to the DHArch understandable execution structure. This means the separation of the description from the execution and it has an advantage. It reduces the complexity of the engine while it is providing a powerful expressiveness for the handler orchestration. With this effort, the engine that carries out the execution according to the internal handler orchestration structure is kept as simple as possible.

The simplicity is an important key feature of software mechanisms. Having powerful, efficient but simple systems has always been tradeoff. It is a challenging issue to weigh among them. However, having enough simplicity without hurting efficiency is the feature being sought in a good design. Therefore, we reduce the burden over the engine by making it as simple as possible while we do not cause inefficiency in the system. In fact, we contribute to the efficiency by forcing simplicity.

One of the important questions for the execution is how a handler orchestration document is converted to the internal orchestration structure. On the one hand, we have four basic constructs which build a handler orchestration document. On the other hand, two execution styles are employed in the execution engine; DHArch contains only sequential and parallel execution in its engine. Hence, sequential and parallel constructs have their exact matches; a sequential construct is mapped into a stages containing only one handler. For example, there will be three stages containing only one handler, if there are three handlers in a sequential execution construct. In contrast, a parallel construct is mapped to only one stage that contains all the handlers. The remaining routing constructs, looping and conditional are converted into these two execution styles. The looping construct is equivalent to the sequential construct comprising of the same handler many times. Therefore, a looping construct is mapped to the structure that has many stages that consist of only the same instance of a handler. There are two reasons for the looping. The first reason is that the nature of the handler may require executions repeatedly. The second reason is the benchmarking; we have utilized looping when we have measured the overhead for the handler distribution. In conditional, the construct is mapped to an execution style that contains the only handlers that pass the conditions.

Since a construct may contain many conditions, one or several handlers can be executed accordingly. Therefore, the mapping can lead either parallel or sequential execution. If several handlers are going to be executed, the execution will be parallel. Otherwise, it is sequential.

Each handler contains a oneway element. It describes that a handler does not have to send a response back. In other words, the flow engine can continue its processing without waiting for the completion of the handler execution. This also affects the parallel construct typeOfParallelExecution element. If anyone of the handlers that are employed by the parallel construct is oneway, the type of construct becomes asynch.

A handler construct contains an element to clarify the action in case of failure. mustPerform is an element containing a boolean variable. It decides the necessity of the execution. If its value is true for a handler, the whole execution must be halted when an error occurs. Otherwise, a necessary action for a Web Service may not have been performed.

Once DHArch interprets the orchestration document, it creates an internal orchestration structure. Every message employs its own orchestration. It basically defines how the message travels through the handlers. Several parameters are utilized to contribute to this effort. Some of them are generated by utilizing the orchestration document while the others are leveraged internally.

4.6 Flexibility and policy schema

Although an internal orchestration structure is initially created by utilizing an instance of the orchestration schema, it is possible to alter the sequence while the execution continues. The modification of an execution is permissible unless the rules

defined in the schema are not ignored. In other words, an individual flow sequence can be assembled for a specific message if the new path does not contradict to the instance of policy schema. However, the modification may not be suitable for some circumstance. Additionally, there might be some other restrictions for the modification even if it is allowable by the schema.

The alteration of the internal orchestration entails additional controlling mechanisms. Even though the adaptability is an excellent feature so that the system offers a flexibility to build individual message flows, the policies should be enforced to apply the limitations and the boundaries to precede the correct flow sequences. Some handlers may process any type of messages arriving to the system without causing any complication. Yet, the others may not be appropriate to be executed without restrictions. There may be a necessity for a compulsory sequence among some handlers. For example, an encryption should be processed at the beginning so that the remaining handlers can understand the message content. Therefore, while the new sequence is created from the available handlers on the fly, the policies have to be kept in mind.

Hence, we come up with another XML Schema to define the policies; see Appendix B. A policy file may contain many descriptions. They define conditions to carry out the execution without having an accident. We choose *any* type for the description element to allow describing any kind of policies. Some definitions may be optional although some others must be compulsory. The schema also defines an important element to describe the orders among the handlers. The policy may comprise of many ordering elements to force the necessary restrictions. Moreover, it contains the

orchestration schema file name and its version to let the system know where the policies need to be applied.

4.7 Summary

Orchestration is a significant feature to collaborate the distributed applications. Handlers are the key components of Web Services. Dissemination of the handlers to have efficient and effective SOAP message execution requires a well-organized orchestration. We introduced the separation of description from the execution in the orchestration mechanism for this purpose.

The separation has many benefits. First of all, it offers very efficient and effective flow engine while it is providing very powerful expressiveness in the description. Weighting between the simplicity and the efficiency is always an issue. Without sacrificing the efficiency, acquiring simplicity is very challenging.

Secondly, the separated mechanism provides an advantage to be able to get support from Petri net model that offers proven mathematical and flow model. The description document can facilitate a visual workflow system for the simulation. Many visual workflow tools have been introduced so far. They can be the supporting tools to analyze the orchestration in many conditions. The orchestration document comprises of enough expressive power to verify and analyze a deployment by using the model. While we are building the orchestration, we care the compatibility feature with the Petri net so that the flow mechanism can be converted to this model. Moreover, applying the orchestration document to Petri net model supports the correctness of the flow structure with the mathematical model. It is very constructive in order to fortify an empirical system with a theoretical approach.

Last but not least, the separated mechanism can help us to be able to build a static, semi-dynamic and dynamic handler distribution mechanism. DHArch utilizes predefined handler setup. Handlers and their sequences are described by an XML document, an instance of DHArch Orchestration Schema. This is a static handler deployment. The orchestration engine interprets the document, creates and executes the sequence.

Moreover, our approach allows to build semi dynamic handler execution mechanism. The handler sequence can be optimized on the fly. The predefined sequence can be altered via introducing parallel execution among the appropriate handlers or rearranging the order. This arrangement must be controlled by a policy document, an instance of DHArch Policy Schema. While the sequence is being altered, the policy document imposes the rules to enforce the modifier to obey the dependencies. Hence, the handler orchestration sequence is modified without breaking the rules defined in the policy document.

Finally, the more appealing but complex mechanism, fully dynamic execution, can be built. The handlers and their sequence are resolved by DHArch. When a message has arrived, the system looks at the context and decides the required handlers and runs them in ad-hoc manner. It can check whether they can be executed parallel or not. It decides which handler should be executed first and so on. This mechanism requires very complicated module, an agent base system.

CHAPTER 5

DISTRIBUTED HANDLER ARCHITECTURE EXECUTION

5.1 Distributing handlers and possible environments

DHArch offers abundant computing resources for the distributed handler execution. It provides an environment to distribute the Web Service handlers to either virtual or physical machines. The handlers can benefit from the utilization of not only individual computers but also virtual machines. If the resources suffice, DHArch is able to benefit from a single machine. Otherwise, it utilizes additional resources to gain the extra computing power.

There exist several scenarios for the handler distribution. The first scenario is the single computer usage. A computer may be a single processor, multiprocessor or multi-core machine. Each of them has its own advantages. We note that the multi-core systems

are very important because the usage of the core in computers is increasing very fast; computers are expected to process many cores in near feature.

One of the best possible hardware configurations is the one benefiting from the utilization of multi-processor or multi-core systems. The effect of message pipelining is discussed in the section 6.3. A distributed handler may exploit an individual processor or core. This provides an excellent environment for the concurrent handler execution. Each handler can acquire its own core or processor and complete its task without sharing the computing resources. Processors and cores are the most important resources because they are the units where the execution happens. Assigning a processing unit to a single task and executing them in a parallel manner may reduce the total processing time.

A single processor computer may not be as good as the multi-core and multi-processor system for the performance wise. By exploiting multiple cores or processors, we are able to remove the limitation over the computing resources. Processing handlers in a parallel manner on a single processor may cause too frequent context switches; we can only exploit the parallel execution for a single processor best in the situation that a handler claims the processor while the other handler are doing their I/O tasks. However, we cannot expect this situation very often.

Multiple machines can be efficiently utilized for the distributed execution. Each handler may acquire an individual computer within a network to contribute to the execution with the additional computing power. Even though there are overheads and obstacles for the distribution and the management of the execution, the use of the additional computer provides very suitable environment for the handlers due to the recent improvements in the network speed, especially in Local Area Network (LAN). However,

the overhead should be compensated by the advantages and gains provided by the utilization of the multiple computers to justify the usage.

Although there seem to be more security issues in LAN than a single machine, LAN usage provides enough protection if a specific LAN set-up is assigned for the distributed handler execution. In other words, we can dedicate a cluster for this purpose. The cluster can be forced to have only one gateway to the outside world so that the unauthorized access points can be limited. Although every computer on this network can join the computation, the communication to the outside is achieved through a single computer. By doing this, the vulnerability points are reduced to a single one. This structure can provide very powerful Web Service environment; one computer hosting the service endpoint and many additional computers helping the execution. The idea is very close to loosely coupled multiprocessor computer systems.

The last scenario is about deploying the handlers over Wide Area Network (WAN). It brings many issues that need to be solved for the handler distribution such as security and reliability. The issues are different than those in the LAN. LAN can be dedicated to a specific purpose and the threats are minimal so that DHArch can utilize it as if it is running in a single machine environment. Since the numbers of the threats by which are exposed in WAN increase dramatically, we may not do the assumptions as it is in LAN.

5.2 Execution

DHArch is a system that is capable of processing Web Service handlers in a distributed environment. It is able to use single-processor, multi-processor or multi-core

systems as well as facilitates multiple computers sharing a network. It supports the parallel as well as the sequential execution.

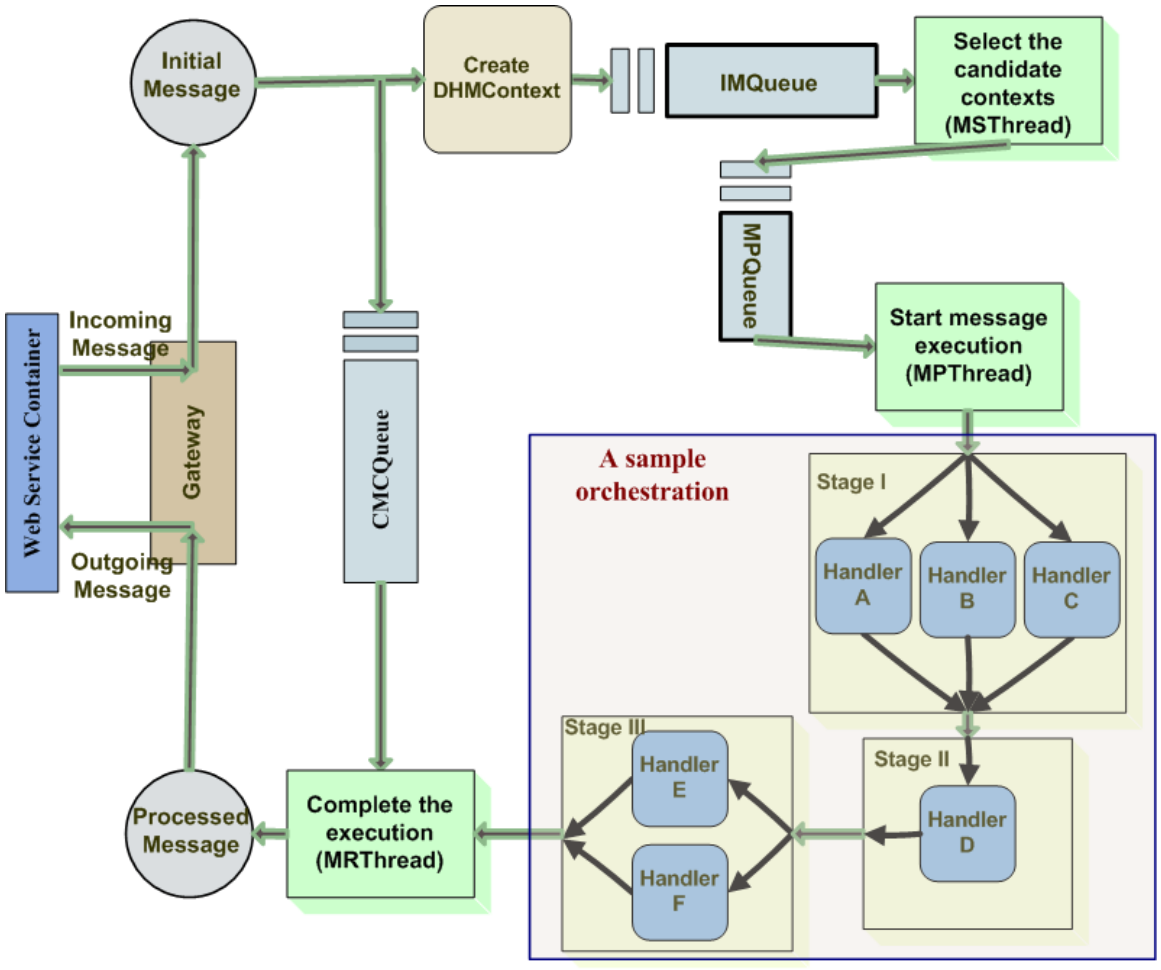


Figure 5-1 : A message execution

DHArch is transparent to the clients. It can be portrayed as a black box; it accepts a message as an input and provides a processed message as an output. A user does not necessarily know how the execution happens in a distributed manner. Simply, a client requests a service and expects a response. DHArch internally processes the arriving request in the distributed handlers.

Figure 5-1 illustrates how a message traversal happens in DHArch. Queues, context objects, XML schemas, parallel and sequential executions and handler orchestration facilitate the execution. We will explain the journey in the remaining part of this chapter.

5.2.1 Message naming

Naming is very vital to identify an object, a product or even a human. Every one of us has a name. With the extensions, last name, birth date and parent's names, we are uniquely identified. We are compelled to use our names to perform our daily activities. Without having a name, we could not even have attended to the schools, worked in a place or built businesses. Shortly, we cannot achieve our current lifestyles without being uniquely identified.

Many messages may arrive to the DHArch in a short duration. The confusion is possible if we cannot differentiate them from each other. The requests can be made from the different sources as well as originates from the same place. Hence, every message has to be uniquely identified to be executed correctly as it happens in many things in our life. Otherwise, the execution cannot go through properly because of having confusion in source, destination or processing.

Therefore, we assign a unique identifier to each message while arriving. A 128-bit UUID generated key is given to every message. The generator assures that the same identifier is not likely to be given to the different messages. This assurance provides enough uniqueness for the message processing.

5.2.2 Message acceptance

In DHArch, the second stop of a message is the acceptance. Typically, a message arrives within a context, specifically Web Service container context. The context consists of additional information for the execution as well as the message. It also conveys supplementary information about the service requester. Therefore, the context object is stored not to lose the necessary information while the execution happens. By using this record, the response to the right client is guaranteed even if many requests are received from many clients.

DHArch stores interacting platform specific context objects without changing the format. This is necessary for the flexibility. DHArch cooperates with the various Web Service containers. Since every container makes use of its own context object for the internal execution, creating a common format for the contexts requires deep knowledge about each one of them. Moreover, conversion between the context objects and DHArch specific common format would be costly. Hence, we decided to keep them as they are. This is reasonable because the interacting container context objects are kept so that the native container can continue its execution after DHArch completes its own.

For the storage, a queue, Container Message Context Queue (CMCQueue), is utilized to save the interacting container contexts. The queue is extendable when it is necessary. Any kind of object is able to be stored without concerning its type. The object is mapped with a unique UUID generated key given to a message when it has arrived.

DHArch creates its unique message context, Distributed Handler Message Context (DHMContext), to perform its internal execution properly. A context object keeps the necessary information about a message. The container contexts are not utilized

for this purpose because of the reason we explained above; we want to build an architecture having a container independent execution. Otherwise, we need to revise the execution mechanism for newly introduced containers.

DHMContexts are also stored in a queue, Incoming Message Queue (IMQueue). Utilizing queues let DHArch accept every arriving message. Otherwise, the messages would have to wait to be accepted or need to be dropped when the message rate is too high. Instead, we choose to accept and offer them the service in an orderly fashion. Similar to CMCQueue, IMQueue identifies DHMContext with the unique identifier.

DHMContext employs several structures and parameters to contribute the execution. The most important one is the handler orchestration structure. It defines the sequence of the handlers. A sequence consists of stages and their corresponding handlers. Moreover, DHMContext contains the message too. The message can be updated while the execution continues. Additionally, several supplementary parameters are kept within the context to facilitate the execution.

An orchestration structure is very important to have an efficient and accurate execution. This structure is generated for every incoming message. The detailed information about the orchestration can be found in CHAPTER 4. DHArch utilizes an orchestration mechanism that separates description from the execution. A basic orchestration descriptive constructs are utilized to address very complex configurations. The engine simplifies this complex orchestration configuration during the execution to reduce the complexity of the management. This allows having easily manageable and very effective execution environment for the distributed handlers.

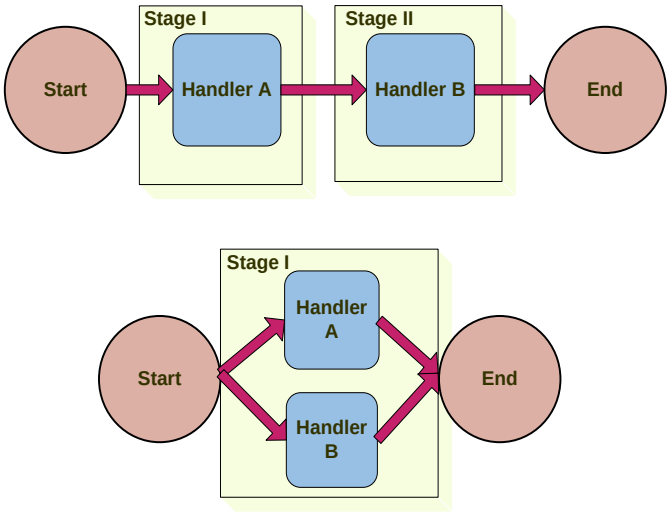


Figure 5-2: Sequential and parallel executions for Handler A and Handler B

All basic orchestration constructs are mapped to two simpler processing styles, sequential and parallel. The important question is how sequential and parallel execution happens. *Stage*s are introduced to support parallel execution. Many stages can be employed in an orchestration structure and their executions are sequential among each other. However, the handlers in a stage are executed concurrently. Each stage should contain at least one handler and there must be more than one handler in a stage to have a parallel execution. Figure 5-2 depicts sequential and parallel executions. If the handlers are in the separate stages, they are executed sequential otherwise they have to be processed concurrently.

The message processing happens based on the guidance of the orchestration structure. Although it is initially loaded from HODocument, the orchestration is not static. It can be modified during the execution if the orchestration policy allows it. The policy contains the laws about *must* and *mustn't*. A handler orchestration structure may contain several conditions for the correct execution. The policy may dictate the execution sequence. For example, an encryption handler is forced to be executed first. Hence, the

conditions have to be followed while the modification of the orchestration structure is happening.

Queues work as regulators when the large number of requests arrives. Accepting the requests arriving to the Web Service and processing them during the appropriate time increase the system responsiveness; in a service/client interaction, a service sometimes does not get a message while it sometimes receives too many messages to handle.

When DHMContext is generated and its insertion to IMQueue is completed, the acceptance of the message is finalized. At this moment, the cooperating container context is safe in CMCQueue and DHMContext objects waiting to be selected for their executions are ready in IMQueue.

5.2.3 Message selection

While DHMContext objects are waiting to be executed in IMQueue, a worker thread, Message Selector Thread (MSThread), starts selecting the candidate messages. The candidates are decided according to the First Come First Serve (FCFS) scheme. It is a fair selection because the first arriving message is chosen to be processed first [92].

However, the selection scheme can be changed to another queuing scheme such as priority. Let's think a scenario that we have a special client so that the requests coming from this client need to be executed right away. In order to provide the necessary privilege to the client's request, we have two options. The first one is to convert the queue into priority queue [93]. The message contexts are inserted into the queue according to their priorities; the high priority message is positioned at the top of the queue. The second solution is delaying the prioritization. The messages are inserted in an order that they arrive. But they keep track a variable that shows their priority for the

execution. When a selector thread is running, it looks at the value of this variable and selects the candidate accordingly. Both of the solutions are valid. If the number of contexts in the queue is high, the first solution is more reasonable.

MSThread chooses the candidate messages and places them into Message Processing Queue (MPQueue). This queue is the place where the parallel message execution, pipelining, happens. There is an optimum value for the number of messages in this queue. Similar management is facilitated in TCP protocol packet rate control procedure[94]. Queue Manager increases the number of contexts in the queue gradually unless the throughput starts diminishing. The optimum value is looked for by increasing and decreasing the number of messages in the queue.

MSThread tries to keep MPQueue full. It checks always whether there exist optimum number of message contexts in the queue. If there are enough messages, the thread sleeps. Otherwise, it selects new candidates from IMQueue.

5.2.4 Sending messages to the distributed handlers

DHArch utilizes messaging for the handler distribution. NaradaBrokering is a very suitable messaging middleware for this purpose. Handlers are able to be distributed efficiently to uniquely addressable places with its utilization. A unique topic is used to identify a handler in its distributed location. In short, the broker works as a postal service that carries the envelopes between the nodes, which have unique P.O. boxes.

The locations of a broker and the nodes are important to reduce the transportation cost. It is ideal to choose the locations to shorten the paths. On the other hand, we may have to place the broker and the nodes far away from each other to utilize the necessary

resources. This is a tradeoff that needs to be dealt with while the decision about the locations is being given.

An execution is initiated when NaradaBrokering is ready to transport the messages to/from the handlers and the messages are waiting in MPQueue for the execution. MPQueue is much smaller IMQueue. We have two reasons to employ a smaller queue. The first reason is the message pipelining. The messages in are being processed concurrently to allow executing more messages at a time. The second reason is to minimize the access time. The idea is similar to the memory structures of the modern computers; the processes are taken into the caches, smaller and faster memory [95]. Similar to this hierarchical memory structure of the contemporary computers [70], DHArch utilizes a smaller dedicated storage, MPQueue, in addition to the bigger one, IMQueue.

The size of the queue directly affects the overall throughput. A single message could have been processed at a time so that we did not have to struggle with the management of MPQueue. However, having this smaller processing queue contributes to throughput very positively.

There are two approaches to manage the queue size. The first one is a static approach; an optimum value is assigned in advance for the size of queue and it is not changed once the execution starts. The second approach is the dynamic management; the size is not fixed. Instead, the queue shrinks and expands in order to keep the optimum number of messages in the queue. The queue length increases to the point that the system performance allows so that the system resources are exploited fully without hurting the performance. If there is not any degradation, the size continually increases. The size is

reduced when the performance gets worse. There are several exemplary managements for this kind of task. We use a simple one that is increasing and decreasing the message number one by one. We may, as well, benefit the concepts from more complex algorithms such as the window management of the TCP protocol [96].

Naively, it can be thought that it would be good idea to use a very large queue. However, we know that the access time increases when the queue length increases. More importantly, processing a tremendously crowded group of messages concurrently depletes the computing resources and causes more frequent context switches. There is a break-even point for the queue size that the performance starts deteriorating while the queue size is increasing. This defines the optimum value of the queue size.

Message Processing Thread (MPThread) starts the execution of the messages in MPQueue at once. It does not stop the processing until the MPQueue becomes empty. While MPThread tries to deplete the messages from MPQueue, MSThread stockpiles new messages on the top of the queue. They work very closely and in tandem style. It is very correct to say that MSThread is a producer while MPThread is consumer. Since we use an instance of *Hashtable* class of JAVA, we do not encounter synchronization problem because it is already a synchronized data structure. Although hashing is a critical issue for queuing performance [97], its overhead is very minimal.

MPThread carries on the message delivery by extracting necessary information from DHMContext. Every distributed handler is located in an addressable place. The addresses are kept within DHMContext. The context also contains the message and the supportive information for the message execution. By using these data, an XML

document, explained in section 3.2.1.5, is created for the transportation. It is an envelope that consists of a unique id, properties and the message itself.

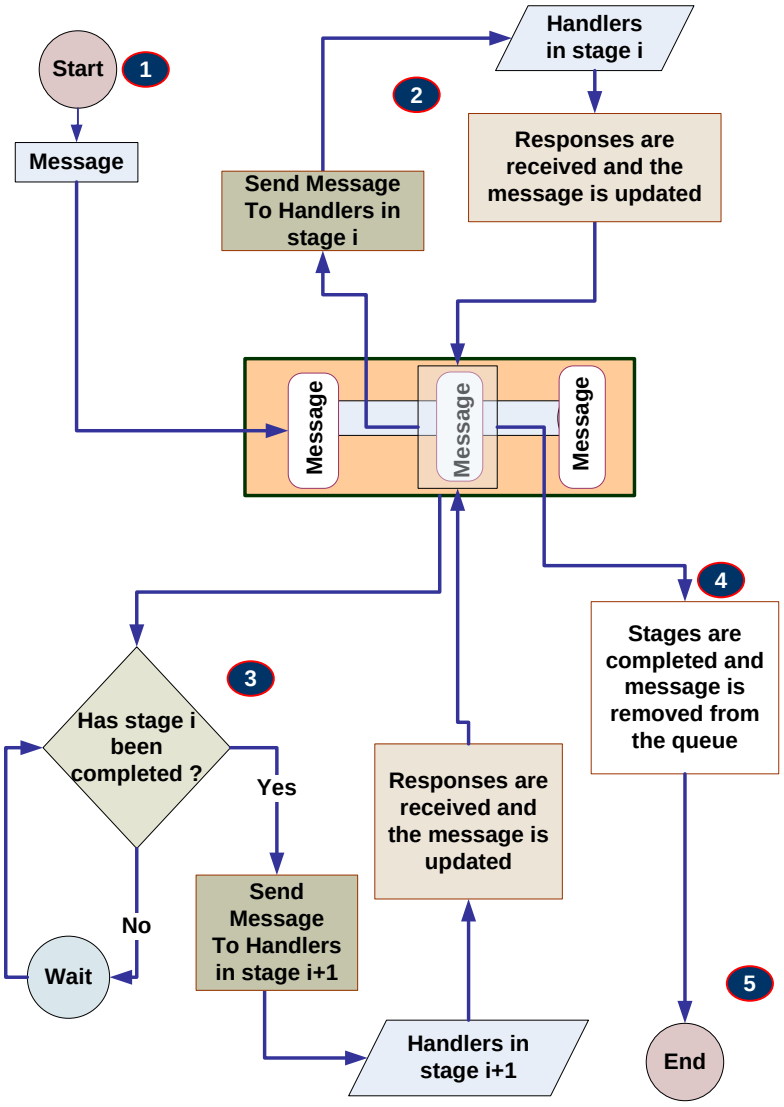


Figure 5-3 : Message execution flow over Message Processing Queue (MPQueue)

When an envelope is ready, it is sent to the distributed handlers with Communication Manager (CManager). We explained CManager in detail in section 3.2.2. Figure 5-3 explains how a message delivery occurs between the stages. The messages

are instantly sent to all handlers of a stage. However, the message execution of those handlers may be completed in different times. All the handler executions in a stage have to be finished before going to the next stage unless the execution is *parallel synchronous*. MPThread waits the completion of the handler executions before starting the delivery of the message to the next stage. This procedure continues until all stages are completed.

Several threads run simultaneously in DHArch. This requires a clever notification mechanism to manage them. They are not allowed to run continuously. Instead, they are forced to wait if they are not needed. Otherwise, wasting the system resources is inevitable. The threads share the computing resources to be successful in their tasks. The resource sharing happens according to the system thread scheduling algorithm. If a thread continues to run with a conditional check instead of staying in its wait condition, it will consume the CPU and memory resources even if it does not perform an actual task [98]. MSThread enters in its wait condition when MPQueue becomes full or IMQueue becomes empty. In both situations, there is really nothing to do for MSThread. Hence, it stays in wait condition until it is notified. There are two notification events for MSThread. The first one is the number of the messages in MPQueue. If the MPQueue becomes empty or contains fewer messages than the optimum number, MSThread is notified. The second notification is a new message arrival. If MSThread and MPThread are somehow waiting in their conditions, they cannot restart their executions because they notify each other. Therefore, an independent notifier is essential to continue the executions. When a new message arrives to IMQueue and the number of the message in MPQueue is less than the optimum value, MSThread receives a notification.

The stateless handlers do not lead to an inconsistent situation when their executions are repeated. Hence, messages are attempted to send more than once to the stateless handlers when an error occurs. In order to achieve this, an approximate handler execution time is kept track. The time can be set initially to an appropriate value. It is updated according to the execution time of the distributed handlers while the execution continues. The modification is slow; the new execution time does not replace the old one directly. Instead, old and current execution times contribute to the new value together. The spikes in a handler execution time that happen because of unexpected situations, such as unprecedented network latency, are eliminated by doing so. Distributed handlers acknowledge the completion of the tasks in the time interval that the approximate time value defines. Otherwise, MPThread sends the message again to that failed handler. This procedure can be repeated for a given number of times otherwise an exception is thrown. This exception is propagated back to the requester to show that the execution cannot be possibly completed.

Even though there may be many distributed handlers in the system in a moment, the message is sent only to those handlers described in the message orchestration document, DHMContext. DHArch may have a handler pool contains many handlers even if they all are not utilized in an orchestration. An orchestration can choose a set of handlers although another one can utilize another set of handlers. This approach suits best for dynamic handler execution.

5.2.5 Message processing in the distributed handlers

Handler invocation occurs according to the DHMContext orchestration structure. Communication Manager (CManager) delivers the messages to their destinations in an order defined by the context.

When a message is received by Handler Execution Manager (HEManager) via CManager, the preparation of the suitable environment for the execution is initiated. A message arrives in an envelope, DHArch Messaging Format. Therefore, the essential information for the execution needs to be extracted from the context. The envelope includes the unique ID, properties and payload. The payload and properties contributes to the handler execution. The unique id is necessary to identify the message and to be able to send the response back. It is very crucial for the correctness of the execution.

DHArch can utilize wide variety of handlers such as monitoring, format converters, logging, compression, decompression, security, reliability and so on. They generally perform tasks that support to Web Service by introducing a new functionality. The interesting part of a SOAP message for a handler is the header even though the body is able to be processed. Therefore, a handler mostly expects the whole SOAP message as an input. On the other hand, many handlers process only the partial SOAP messages. For example, WS-ReliableMessaging handler processes only *wsrm* tag of the entire message. Therefore, HEManager allows utilizing the partial execution where the size of the message becomes a concern. However, since this is not applicable to every handler, a full SOAP message execution is performed unless the partial execution is explicitly mentioned as a necessity. We also need to keep in mind that the partial SOAP message

execution causes an overhead originating from parsing the SOAP message and combining the outputs later.

HEManager exploits supplementary data for the handler executions. These data are conveyed within the properties. Some of these properties are applicable to every handler. One of them is oneway feature. It describes a situation that a handler does not have to send any response back. *oneway* property is in the scope of both DHManager and HEManager . Therefore, when DHManager encounter an *oneway* handler, it applies fire and forget paradigm and continues its remaining tasks without waiting the response [99]. On the other side, HEManager does not waste its precious time with an unnecessary task. This policy improves the throughput of the overall system for the appropriate handlers.

Additionally, *mustPerform* property is also universal for the handlers. If a handler has true value for the *mustPerform* parameter, it always has to complete its executions. In the situation of an error, the execution has to be repeated if it does not lead to an inconstant state. Otherwise, the message execution must totally be halted and the requester must be informed. The message execution can continue when the *mustPerform* value is false even if the handler throws an exception. For example, skipping a logging handler may not be so crucial for a Web Service so that the message execution can carry on without restarting it from the beginning.

5.3 Getting response back

When a handler completes its task, the output message is pushed back to the HEManager. This output has to be wrapped with the same envelope that request message has arrived, DHArch Messaging Format (DMFormat). The corresponding unique ID has

to be used in this envelope. When it is ready, it is delivered to CManager for the delivery to the destination.

CManager forwards the envelope to its address. When the envelope arrives to the destination, another thread, Message Response Thread (MRThread) is activated. The message delivery is a notification to MRThread. It updates the corresponding context with the executed message. First, it checks whether the ID is represented in MPQueue. Otherwise, the response is behaved as a malicious message and it is discarded. If the ID passes the check, the properties and the payload are extracted. The corresponding DHMContext in MPQueue is retrieved by using the unique ID. At the end, the context is updated with the processed message.

The modification of a context with a successful handler execution may not be the end of the journey. The message has to repeat these procedures for every handler in its orchestration. MRThread checks whether the message completes the execution for every handler. If it is the case, the context is taken out from the queue.

The container context object has been kept in CMCQueue until this moment. It was preserving the essential information to continue the message execution in the interacting Web Service container. Therefore, saving the container context object is very important. When the container context is taken out from the CMCQueue, it is updated by utilizing DHMContext that we have retrieved from MPQueue. Finally, the processed container context is passed back to the Web Service container to finalize the message execution in DHArch.

5.4 Error handling for distributed handlers

DHArch provides an environment for the distributed handler execution. It is possible to have errors while the execution is happening. If a handler stops abruptly because of a failure, the error need to be handled so that the system continues to its execution. An error is a state that may lead to a failure. The reason of an error is called as fault [100]. Being clear about the basis of an error is crucial to provide a solution. Laprie et al. describes two ways of dealing with failures, *fault prevention* and *fault tolerance* [101]. While the first one works to prevent the occurrence of a fault, the second copes with providing the continuation of the service even in the presence of the failure. Even though a complete avoidance of failure is not possible, there are tools supporting fault prevention [102].

Apparently, fault tolerance is necessary to be able to continue execution while a fault occurs. Fault tolerance requires enhancing the language to detect and handle the error. Additionally, a new semantics is essential to modify the execution on the fly[103] .

When a fault tolerance is mentioned, we need to bear in mind that forward recovery can be used as well as the backward recovery. In the forward recovery, the tasks are tried to be completed by processing several times.

Backward capability requires atomicity. It is one of the most essential notions for the consistency. In regard to atomicity, Hagen at al. [103] defines three task types, *atomic*, *quasi-atomic* and *nonatomic*. Atomic tasks are those that they have no effect at all if they fail. For example, every read-only task can be thought as an atomic task even if they fail because it does not cause any change. *Quasi-atomic* effects do not vanish

naturally. The effects can be eliminated via a roll-back action, though. *Nonatomic* tasks are the one that the effects cannot be removed when they are committed.

Handlers can be either statefull or stateless. A handler generally processes a SOAP message and applies its procedure over it. In other words, they do not keep any state for the message. This feature contributes to utilizing forward recovery. DHArch restarts the execution if a stateless handler fails. HEManeger notifies the error to DHManager. In other words, the exception is propagated back to DHManager. DHManager starts the message execution again when it receives the exception for the stateless handler. It may be repeated several times depending on the situation. If the execution is not successful after these efforts, the message execution is totally halted and the exception is propagated back all the way to the service requester. In this case, the requester may assume that the handler may be down or crashed.

Handlers are not always stateless. They might be keeping states for the messages. DHArch expects atomicity from the statefull handlers. If a handler fails during its execution, it should not have any effect at all. If having atomic handler is not possible or the handler is a *quasi-atomic,* it is necessary to utilize two-phase commit. There exists a solution for the distributed commit [104]. However, we prefer to employ a handler in a suitable place to commit or roll-back the effects if the handler is not atomic and statefull.

There exist cases that the execution can continue even if an error occurs. The handler orchestration consists of a property that defines whether it is an obligatory to be performed. *mustPerform* element tells to the system whether it has to be executed. If a handler contains true value for *mustPerform*, the message execution cannot continue

without achieving its execution. Otherwise, the error can be neglected and the execution continues.

5.5 Management of handler replicas

Replication is critical to mobility, availability, and performance of a computing system. We benefit from the replication in our daily life too. Even our body benefits from the replications; we have two legs, hands, eyes and ears. We keep a spare tire in our cars to replace a flat one in an emergency. The important files are backed up to reduce the probability of lost. Computing systems also utilizes the same strategy via replicating the data and the computing nodes.

There are basically three replications: data, process and message. These concepts are extensively explored [105]. Data replication is the most heavily investigated one. However, the other replications are also very important in the distributed systems, especially for Service Oriented Architectures.

We are particularly interested the process replication because our intention is to investigate the replication of the handlers. Process replication has been mentioned in the literature even earlier than the data replication [106]. There exist two main approaches in this area. The first one is *modular redundancy* [107]. The second approach is called *primary/standby* [108]. *Modular redundancy* has the replicated components that perform the same functionalities. All the replicas are active. On the other hand, *primary/standby* approach utilizes a primary replica to perform the execution. The other replicas remain in their standby state. They become active when the primary replica fails.

The processes can be classified in two categories; no consistency and consistency. The fist category is the simplest one. The processes are stateless. They do not keep any

information for the processed data. Therefore, the consistency is not an issue between the processes. Replicated instances can be allowed running concurrently. On the other hand, replicas may enter in an inconsistent state if the process is not atomic and statefull. Inconsistent processes has been extensively investigated [109].

Replication is a very important capability where the handlers are inadequate. Sometimes, a handler may not be sufficient to answer the incoming requests. The tasks may line up so that the overall performance degrades. This is similar to a shopping center where the customers are waiting in the line to be served. Let's assume that customers are served by one person and that person is able to serve one customer in a minute. If two customers arrive in one minute, the number of costumer will increase in every passing moment. The solution is to add one more person to serve to be adequate for the customers. Similarly, adding a handler to help the execution contributes the overall performance.

Additionally, a replica can be leveraged when an error occurs. We explained the error handling in the previous section. It is possible that a handler crashes. We may utilize a replica of a handler when the primary one is unavailable. This solution contributes to the continuity of the execution and improves availability of the service in the overall system. This contributes as a recovery mechanism when an error happens during the execution[110].

We utilize a variation of *primary/standby* approach. The replicas are prioritized. The handler having highest priority is selected to execute the message. The other replicas wait until their priorities became highest. The system is able to change the priority during the execution. We never allow the replicas being executed concurrently unless they are

the instance of the stateless handlers. Even though they are allowed to run in parallel manner, they cannot process the same message. The messages have to be different so that the parallel execution does not cause inconsistency.

5.6 Security

Security is one of the important issues for the computing systems. The very critical data can be seen or altered by an unauthorized person. This is increasingly important if the data is transferred through the network, which is more vulnerable environment.

The local computing is not exposing its data to the outside world very much. In contrast, this is not the case for the distributed computing. The computation is shared between the nodes which may physically disperse in the distributed computing. The transmission of the data among the nodes may expose the critical information to the dangerous vulnerabilities. Hence, the transportation channels must also be secured in addition to the security of the computing entities. We will discuss local and wide area network security solutions in the following paragraphs.

Local Area Network utilizes Ethernet technologies. Most of the efforts in LAN have been concentrated on providing secure network gateways. Generally, the private external communication is encrypted and firewalls are utilized to secure internal access.

Unfortunately, there exist several security threats stemmed from Ethernet [111]:

- The single physical line is shared by all the stations to communicate each other. This can cause an eavesdropping of packages by an attacker because every packet in the network can be seen by anybody which is connected to the network.

- There is no way to authenticate the message originator in the Ethernet technology. This can cause a malicious user can insert a modified packages to the network.

Although there are extra security issues, network usage should have the same level of protection as if the local resources are utilized. One way of achieving this goal is to build a specific configuration. LAN Network can be forced to have only one gateway to the outside world so that the unauthorized access points can be minimized. Although every computer on the network can join the computation, the communication to the outside world is achieved through a computer. This structure can provide very powerful Web Service environment; one computer hosting the service end-point and many additional computers contribute to the execution. The idea is very close to loosely coupled multiprocessor computer systems. Computers execute a process by sharing the tasks among each other in loosely coupled multiprocessor systems [112].

Second solution is in the hardware level. The week points originated from Ethernet can be removed by applying several cautious steps. Each node can read the packages that are addressed to that node. Additionally, nodes can be forced to read a package only once. Finally, each node can verify the originator of data. However, if we handle these steps in software level, it will be costly and it may require the additional protocol. Instead, we may leverage a secure Ethernet NIC. This device provides both Ethernet functionality and encryption for the communication in one PC card. Every security procedure is transparent to the user applications. Thus, the application level does not get complicated. We can enumerate the benefits as follows [111]:

- No unprotected data can be physically sent

- A unique identifier in each network packet prevents an active attacker from replying packages.
- No additional CPU resources require.
- The cards use the centralized key exchange model.
- The only way to get private key is to tamper hardware.

On the Wide Area Network side, the number of the threats increases. The data on the wire can be sniffed and altered more easily. Many technologies offered to provide security for WAN such as VPN, SSL. These technologies construct secure channels between the nodes and help to build virtual networks. Although they may offer secure environments enough, they may add new costs to the overall system performance. Sometimes, the overhead may become unacceptable; over 100 Mb/s Ethernet link, the transfer speed can degrade more than 65% and CPU usage can reach to 90% level, when a strong encryption is utilized [113].

There are many products which attempts to provide security on hardware level to reduce the burden over the CPUs. These products can create dedicated networks including computers that have special hardware for VPN connections[114]. The connection speed can reach up to 1Gbit/s. Creating VPN in hardware level will not increase complexity of an application and cost CPU resources.

Even though hardware level security seems good choice in terms of performance, we may not always utilize them. The local machines and dedicated LAN environments provides enough security with a reasonable cost. On the other hand, WAN environment may require extra security features.

Unique message ID is a positive contribution for the message authentication. It is a unique name for the messages. An encryption mechanism may provide the necessary security on the top of this authentication in the special environments. Moreover, NaradaBrokering has a security framework that is able to supports secure interactions between the distributed handlers with a reasonable cost[43] .

5.7 Reliability

Reliability is one of the most important issues in the distributed systems. DHArch benefits from two different sources for the reliability, NaradaBrokering and its own mechanism. The messaging system, NaradaBrokering, provides a message level reliability. The messages can be queued up to 10000 messages and are gradually delivered to their destinations. Additionally, NaradaBrokering has *Reliable Delivery Service(RDS)* component that delivers payload even if a node fails [115].

A reliable mechanism, additionally, is built in addition to the reliability that NaradaBrokering provides. DHArch is able to repeat a specific handler execution in the situation of a failure. Failure decision is made when the response is not received from a distributed handler. There can be several reasons behind being unsuccessful for getting a response. The communication link may be broken as well as the handler may not successfully process the message because of either an error or crash. DHManager checks the possibilities by sending the message several times. In each attempt, it waits for a specific amount of time. This duration is either assigned or calculated by the system. After having several unsuccessful attempts, the message processing is switched to a replica if it exists. As we discuss previously, handlers may have their replicas to improve availability.

5.8 Summary and conclusion

DHArch provides an environment to distribute Web Service handlers. Handlers can be executed in a parallel manner as well as sequentially even though the conventional Web Service processing environments do not benefit from it. The handler parallelism depends on the handler nature and the possible deployment configuration. Some handlers need to be executed lonely. Some handlers should be processed either before or after a specific handler. In the worst case, every handler should be sequential. The best scenario is able to process the handlers in a parallel manner. A common deployment configuration, at least, lets some of the handlers run concurrently.

Two kinds of concurrency are utilized by DHArch in an execution; handler parallelism and message pipelining. These features provides very efficient environment to the handler executions. Parallelism theoretically removes the barrier in front of having the best performance.

DHArch provides several architectural benefits even if a performance gaining is not possible. The system becomes more modular via handler distribution. Handlers become more independent from the service endpoint in terms of execution, deployment, and implementation and so on. Every handler can independently be modified. The handler usability is also improved by the distribution. Either client or service can access to a handler in their both request and response paths. Only one single handler may suffice for the entire system. DHArch can also let the multiple services access to a handler. However, we need to be careful not to make that handler a bottleneck.

DHArch may contribute the overall system by removing the bottlenecks via replicating the handlers. Some handlers may require so much processing time that they

may cause convoy effect for the arriving messages. By introducing a replica, we may increase the responsiveness of the system.

Scalability is very important criterion for the distributed systems. Because of the utilizing additional resources, the scalability of the distributed handlers improves wonderfully. The usage of powerful machines or the distribution of them among multiple cores or processos causes that the system scales very well.

NaradaBrokering is a positive contribution. It is a reliable, secure and proven messaging middleware. Messaging perfectly supports seamless communication, a key feature of building interoperable systems. NaradaBrokerig provides in-order delivery mechanism in addition to having guaranteed delivery feature. It may even support context base message delivery. Additionally, it can be utilized as a queuing system that regulates the message flow. Depending on the message size, it can preserve as many messages as possible for the delivery. In a benchmark result, we witness that 10000 small messages can be regulated in a moment in one broker.

CHAPTER 6

MEASUREMENTS AND ANALYSIS

We performed extensive series of the measurements illustrating the advantages of DHArch in various environments. The first set of measurements is to examine the performance of a single message in DHArch within various hardware configurations. The second set of measurements is to figure out the overhead of a handler distribution. The third set of experiments is conducted to explore the scalability to illustrate the efficiency of the system. Finally we perform measurements by using two well-known Web Service Specifications, WS-Eventing and WS Resource Framework.

6.1 Performance measurements

DHArch offers a promising environment for Web Service handlers. It supports concurrent execution and allows utilizing additional resources. There can be many types of resources such as computer, processor, memory, storage or even an application.

Although DHArch improves the system performance because of the parallelism and additional resources, the management of these components may cause overhead. Hence, we will investigate the system performance in the remainder of this section.

6.1.1 Handler configurations

DHArch is evaluated by utilizing 6 different configurations of 5 Web Service handlers. The results are gathered by using Apache Axis version 1.x. An XML based WSDD configuration file describes the handler execution sequence in Apache Axis. It supports only sequential handler execution. On the other hand, DHArch provides more flexibility for the handler deployments; it does not restrict the execution with the sequential style. Instead, it supports parallel handler execution too. The combinations of the parallel executable handlers can create so many different configurations. However, the dependencies between handlers and the performance issues need to be carefully investigated for the correctness of the execution while deciding these combinations. 6 different handler configurations are selected for the experimental purpose.

5 handlers are utilized in these measurements. They are customized for benchmarking purposes. Two of them are CPU bound handlers. These handlers are suitable to simulate CPU-bound applications because the execution time can be assigned by changing an input value. The remaining three handlers have been chosen from the applications that are gradually switching from CPU bound to I/O bound. The first one utilizes Document Object Model (DOM) parser. DOM parser converts a SOAP message to a DOM object and allows walking through the elements. When a SOAP message is received by the handler, it creates its DOM structure, modifies its elements, attributes or tags and returns the modified message as an output. The fourth handler is a more I/O

bound application. Similarly, it parses SOAP messages either by creating its DOM object or by utilizing a SAX parser. The partial or whole message is written to a file. Finally, the last handler receives the message and logs the data into a file and prints out the information about the message. The handlers are named in this performance benchmarks as in Table 6-1.

Table 6-1 : Handler list for the performance benchmarking

Handler Name	Handler Type
Handler A	CPU Bound
Handler B	CPU Bound
Handler C	IO Bound
Handler D	IO Bound
Handler E	CPU/IO

Six different configurations are created from these 5 handlers. The first configuration, shown in Figure 6-1, is deployed in a single computer to gather the results from the handler execution mechanism of Apache Axis 1.x. Since Apache Axis does not allow a parallel handler execution, only a sequential configuration is created.

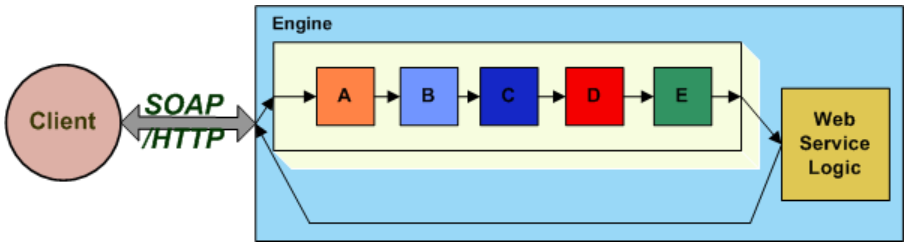


Figure 6-1: Sequential handler configuration in Apache Axis

The same sequential execution is imitated in DHArch handler execution environment. However, there is an important difference; each handler is distributed to the individual computers, illustrated in Figure 6-2.

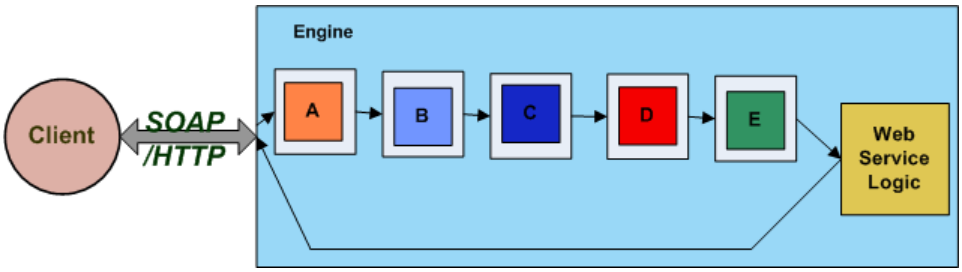


Figure 6-2 : Sequential handler configuration in DHArch

DHArch utilize *stage* concept for the handler execution. A stage is an abstraction to describe a parallel execution for a group of handlers. For a sequential execution, a special case of parallel execution, every stage consists of a single handler. On the other hand, the number of handler in a stage has to be more than one to execute the handlers concurrently. Figure 6-3 depicts a configuration containing three stages and two parallelisms with two handlers.

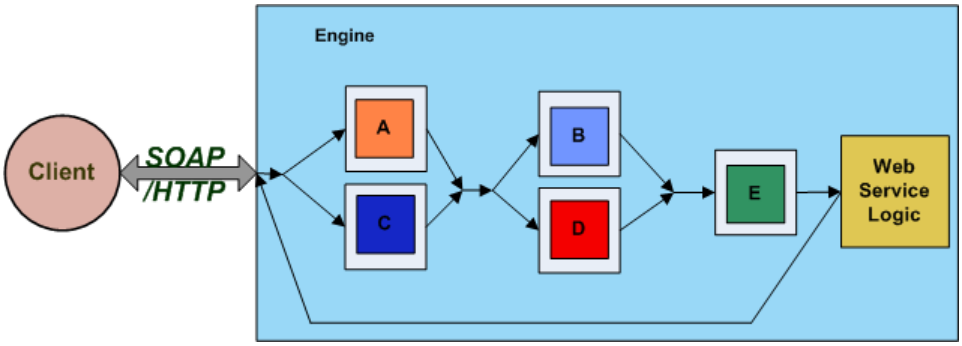


Figure 6-3 : A handler configuration with 3 stages

The execution time of a handler joining to a parallel execution is significant for the performance. The handler configuration, depicted in Figure 6-4, contains the same number of stages. However, the handlers in the first and second stages are different. Therefore, we expect that their performance will be different even though they do the same job.

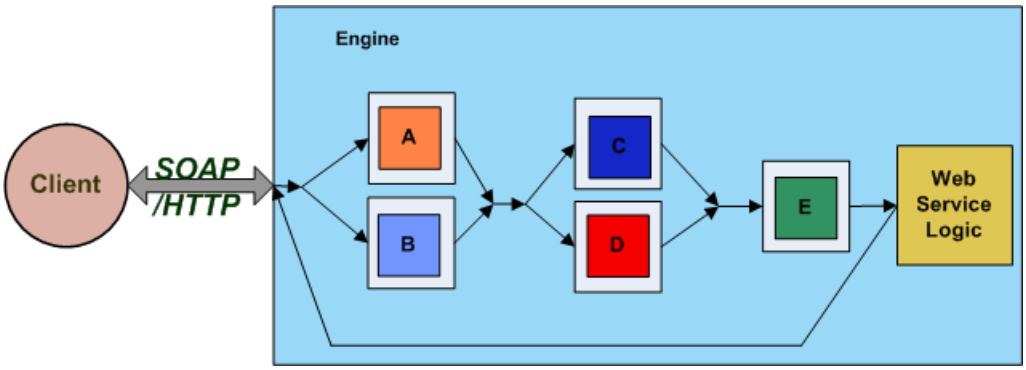


Figure 6-4 : A handler configuration with 3 stages

Figure 6-5 illustrates a configuration with two stages. The handlers are executed concurrently except handler E because of the dependency. It modifies the incoming SOAP message. Therefore, it is kept last to prevent an incorrect execution.

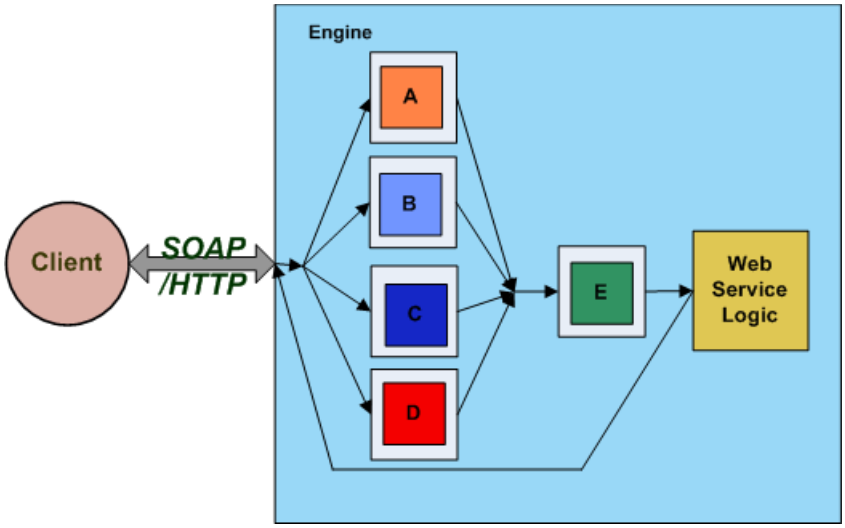


Figure 6-5 : A handler configuration with 2 stages

Finally, a handler configuration, depicted in Figure 6-6, is created without concerning about the dependencies. It consists of a single stage that contains 5 handlers.

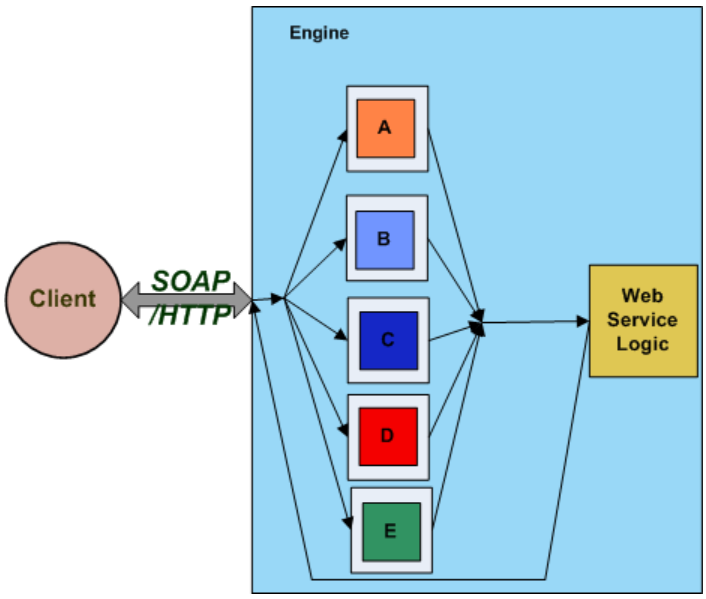


Figure 6-6 : A handler configuration with 1 stage

6.1.2 Environment

The performance experiment is conducted in four different hardware environments. The first environment is a multi-core system. The intention is to figure out the behavior of DHArch in a multi-core system. Nowadays, the trend is to have multi-core computers and it is expected that more cores will be seen in the processors in near future [116]. Hence, we give a special attention to the measurements in multi-core systems. The utilized machine in this experiment has UltraSPARC T1 processor that contains 8 cores running Solaris Operating System, 4 threads per core, with 8GB physical memory.

Unlike Apache Axis, DHArch benefits handler parallelism. Although concurrent execution has many challenges [117], it activates the individual core usage in the multi-core systems; a handler may claim its own core. We can conceive this core acquisition as if every handler has its own computing node so that the tasks are achieved without competing for the computing power.

The second environment is a multiprocessor system, Sun Fire V880, has Solaris 9 Operating System which is equipped with 8 UltraSPARC III processors operating at 1200 MHz with 16 GB Memory. The intention in this system was to figure out the effect of multiple processor system usage for DHArch. Like a multi-core system, operating system may allocate a handler to an individual processor to increase the system throughput. This assignment is valuable when several handlers are running concurrently.

The third benchmarking environment is the computers sharing a Local Area Network. The computers in this cluster have the same hardware features. They utilizes Fedora Core release 1 (Yarrow) in Intel Xeon CPU running on 2.40GHz and 2GB memory. In this environment, the handlers are distributed to the different machines. In other words, each handler has its own computer.

The last environment is a single computer, utilizing Pentium 4 processor operating at 2.80GHz with 1.5 GB memory. It is running Red Hat Enterprise Linux AS 4 operating system. In contrast to previous systems, the distributed handlers need to share a single computing resource. Therefore, we may witness context switches among the distributed handlers and the other components of DHArch that the result may be undesirable for the performance.

Java 2 standard edition version 1.4.2_10 is used in these benchmarks. Web Services are deployed by using Apache Tomcat version 5.5.20.

6.1.3 Individual execution times of handlers

The execution times of the handlers are individually measured to monitor the changes in the different environments and conditions. Each measurement is observed 100 times. The results show the bare processing time of the handlers; the distribution

overhead is sorted out. Table 6-2 shows the results from Apache Axis handler deployment that utilizes sequential execution in the multi-core system. CPU-bound handlers are heavily dependent on the CPU frequency and they have the longest execution times among the handlers. The results do not deviate unreasonably. The standard deviations of the execution times are in the acceptable range.

Table 6-2 : Individual handler execution times in Apache Axis for the multi-core system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	4145.2	19.71
Handler B	2875.8	20.51
Handler C	24.6	5.36
Handler D	50.8	13.08
Handler E	59.4	9.44

Table 6-3 illustrates the measurements of the distributed handlers in DHArch. Since it can execute handlers concurrently as well as sequentially, the results are collected in both execution styles. The processing times do not change in the parallel and sequential executions. The reason is to utilize an individual core for each handler. Since the cores in this specific environment are sufficient to process the handlers even for the concurrent execution, the processing times do not noticeably differ.

Table 6-3 : Individual handler execution times in DHArch for the multi-core system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	4139.41	31.13
Handler B	2893.08	39.15
Handler C	22.33	7.48
Handler D	52.91	17.11
Handler E	58.58	16.80

The results in DHArch and Apache Axis are very close to each other. Deploying handlers into different environments does not significantly affect the processing duration of the handlers in the multi-core system. The reason is that parallel and sequential handlers acquire their individual cores so that the context switches do not occur very frequently. This prevents the deterioration of the results in parallel execution. Therefore, we do not observe a difference between DHArch and Apache Axis executions. Similarly, the results do not vary in an unacceptable range for the standard deviation in both DHArch and Apache Axis environments.

Table 6-4 : Individual handler execution times in Apache Axis for the multiprocessor system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	2044.74	42.66
Handler B	1823.93	18.66
Handler C	21.41	7.78
Handler D	40.54	15.11
Handler E	55.96	14.17

The multiprocessor system yields shorter execution times than the multi-core system because of the faster processors. The processor speeds have been provided in section 6.1.2. Table 6-4 depicts the results of Apache Axis for the multiprocessor system. Even though the execution times are shorter than those in the multi-core system, the standard deviations do not similarly improve. However, they are reasonable in their ranges.

Table 6-5 : Individual handler execution times in DHArch for the multiprocessor system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	2049.2	44.42

Handler B	1831.8	20.50
Handler C	18.6	5.75
Handler D	45.76	12.38
Handler E	49.6	8.57

Since the system computing resources are sufficient enough for both sequential and parallel handler executions, DHArch and Apache Axis handler processing times do not vary significantly. The processing times in DHArch, shown in Table 6-5, are almost equal to those in Apache Axis. Moreover, the results do not fluctuate unreasonably similar to the results of the Apache Axis.

Table 6-6 : Individual handler execution times in Apache Axis for a cluster utilizing Local Area Network

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	1033.64	36.99
Handler B	562.45	22.04
Handler C	16.83	3.06
Handler D	38.90	7.53
Handler E	35.64	7.11

Table 6-6 consists of the processing times of the handlers gathered from a computer sharing a Local Area Network with several other computers. Since Apache Axis handler mechanism cannot benefit from the additional computers, the results are from a single computer. The information about this cluster has been provided in section 6.1.2. The handlers are running faster than the previous systems because of the processor speed. The real speedup is observed in CPU bound applications. The I/O bound handlers are not affected significantly.

Table 6-7: Individual handler execution times in DHArch for a cluster utilizing Local Area Network

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	1031.54	30.51
Handler B	560.54	23.40
Handler C	16.54	2.94
Handler D	32.45	11.31
Handler E	36.29	7.67

DHArch results, shown in Table 6-7, are very close to those from Apache Axis even though each handler is assigned to an individual computer in DHArch handler execution environment. It can be thought that additional computer usage in DHArch provides better performance. However, the handlers in Apache Axis run sequentially. Therefore, we do not observe a worsening in the execution times originating from the context switches because sequential execution does not cause them. On the other hand, the execution times do not also deteriorate while the concurrent execution happens in DHArch because each handler uses an individual computer.

Table 6-8 : Individual handler execution times in Apache Axis for the single processor system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	920.25	22.40
Handler B	498.54	14.58
Handler C	8.83	2.04
Handler D	19.32	1.8
Handler E	26.16	2.94

Finally, a computer containing a single processor is selected to gather the processing times for the handlers. Table 6-8 shows the results from Apache Axis. The

processing times reduced significantly because the computer has the faster processor than the previous systems.

Table 6-9 : Individual handler execution times in DHArch for the single processor system

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	1037.16	21.15
Handler B	517.22	14.73
Handler C	8.67	2.10
Handler D	22.06	4.35
Handler E	27.96	7.87

Table 6-9 shows the sequential handler execution results for DHArch. In previous three environments, the processing times of the sequential executions in DHArch and Apache Axis have been very close to each other. However, we observe that DHArch spends more time to execute the same handlers in this single processor system. It is because of the usage of NaradaBrokering. The number of context switches increases significantly. Therefore, the processing times are slightly higher than those in Apache Axis.

When the parallelism is introduced to a set of handlers, the processing times of the handlers increase more than the sequential execution. The main cause is that the context switches become more frequent because of the parallelism in the handler executions. Parallel handlers start competing with each other to acquire a single processor. Hence, the context switches worsen the processing times although the overall performance improves, explained in section 6.1.4. Table 6-10 provides the processing times for the handler executed concurrently.

Table 6-10 : Individual handler execution times in DHArch for the single processor system while the handlers are being executed concurrently

Handler Name	Execution time (milliseconds)	Standard Deviation (milliseconds)
Handler A	1303.32	86.03
Handler B	538.03	4.40
Handler C	10.06	3.30
Handler D	34.83	7.1
Handler E	44.25	12.59

After presenting the processing times of the handlers in different hardware environments, we will explore the overall performance of the service deployments in the next section.

6.1.4 Overall performance comparison for sequential and parallel execution

We measure the overall performance of a Web Service deployment in Apache Axis and DHArch. Handler distribution causes an overhead. However, there are also gains because of its offerings. In this section, our interest is to find out the performance benefits coming from the advantages of the distribution.

Apparently, the management of the distributed handler execution and the transportation of the tasks affect the execution time. The cost is inevitable but its burden can be reduced by reshuffling the configuration of the execution. In short, there will be always a cost, originated from the distribution, to utilize DHArch.

On the other hand, there are ways of compensating the overhead and even achieving a promising overall performance. The first way of improving the performance of a deployment is to establish concurrent handler execution in a distributed environment.

Apache Axis conventional handler deployment does not let the handlers run in a parallel manner. However, there are many independent handlers from each other so that they can process the SOAP messages concurrently. For instance, a monitoring handler does not depend on a logging handler. They can be easily executable concurrently. The second way of improving performance is to utilize faster machines. From the results of previous section, it is deducible that the time spending for a handler differs from computer to computer. A faster machine may contribute to the overall performance when an appropriate handler is deployed into it. For instance, encryption and decryption handlers' distribution to the faster machines within a secure environment contributes best to the overall system.

The measurements shown in Figure 6-7 depicts the results from the multi-core system, explained in section 6.1.2. The values show the round trip time of a service request. Clients record the time of the request initiations and calculate the elapsed time when they receive the responses. Hence, the measurements contain transportation, service and execution times of the handlers. Every observation was repeated 100 times.

It is clearly seen that the best results are observed when the handlers are able to run concurrently. However, processing them concurrently may not be always possible. As we discussed earlier, the rules between the handlers have to be obeyed; the dependencies have to be considered. For example, a security handler needs to be processed first. Otherwise, the remaining handlers cannot understand the message because of the encryption.

The difference between configuration 1 and 2 is the overhead originating from the distribution of 5 handlers. The first configuration is to utilize Apache Axis in-memory

handler deployment. The second configuration is to deploy the handlers to the individual cores in a single machine by using DHArch. Apache Axis deployment is the reference point for the comparisons. In order to make an accurate comparison, configuration 1 and configuration 2 utilizing Apache Axis and DHArch respectively are same: sequential execution. Because of the distribution of the handlers to the individual cores, DHArch increases the execution time slightly.

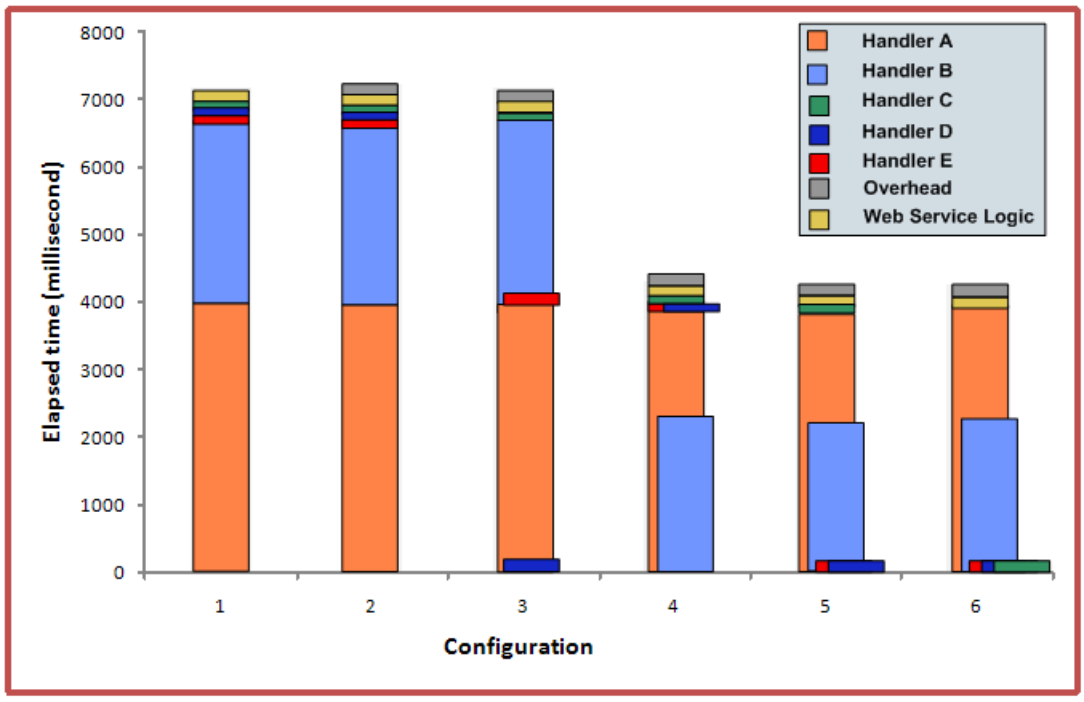


Figure 6-7 : The service execution times of the six handler configurations containing the five handlers in the multi-core system

Parallel handler execution reduces the overall execution time. The gain may be small; in the configuration 3, it is around 50-70 milliseconds because of the total processing time of Handler C and Handler D. As a result, this configuration slightly provides enough gain to overcome the overhead. Sometimes, gain may not even compensate the overhead. On the other hand, a gain can be very appealing. For example, parallel executions in configuration 4, 5 and 6 provide good results due to processing

times of Handler A and Handler B. Their execution times are enough to improve performance considerably. The numerical values of the results are stated in Table 6-11.

Table 6-11 : The elapsed time for the service execution and the standard deviation of the performance benchmark in the multi-core system

Configuration number	1	2	3	4	5	6
Mean value (msec)	7192.9	7220.92	7164.98	4324.86	4279.37	4264.78
Standard deviation	42.97	56.68	57.75	49.66	29.92	36.96

There is a limit for the performance gain coming from the concurrency. We cannot shorten the total handler processing time more than the longest handler execution time. For example, all the handlers may not possibly be processed within the duration of time that is less than Handler A‘s execution time even if all the handlers are processed concurrently.

A percentage of a gain completely depends on handler configuration. On the one hand, it can provide a fascinating performance with the execution of all the handlers in a parallel manner. On the other hand, it cannot even present a gain to compensate the overhead coming from the distribution of handlers.

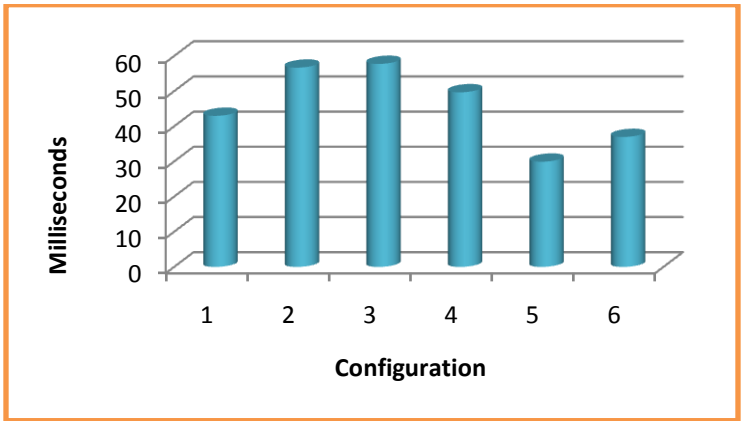


Figure 6-8 : Standard deviations of the service execution times in the multi-core system

Figure 6-8 depicts the standard deviations of the handler configurations. The deviations are reasonable because the executions times vary between 4000 milliseconds and 7000 milliseconds; around 50 millisecond deviations are acceptable.

Figure 6-9 depicts the results from the multiprocessor system, explained in section 6.1.2. The pattern is similar to that observed in the multiprocessor system. DHArch sequential deployment which replicates the same sequence of the Apache Axis handler deployment has higher processing time than that from Apache Axis due to the overhead of the handler distribution. In general, the gain does not only compensate the overhead but it also shows very significant performance improvements.

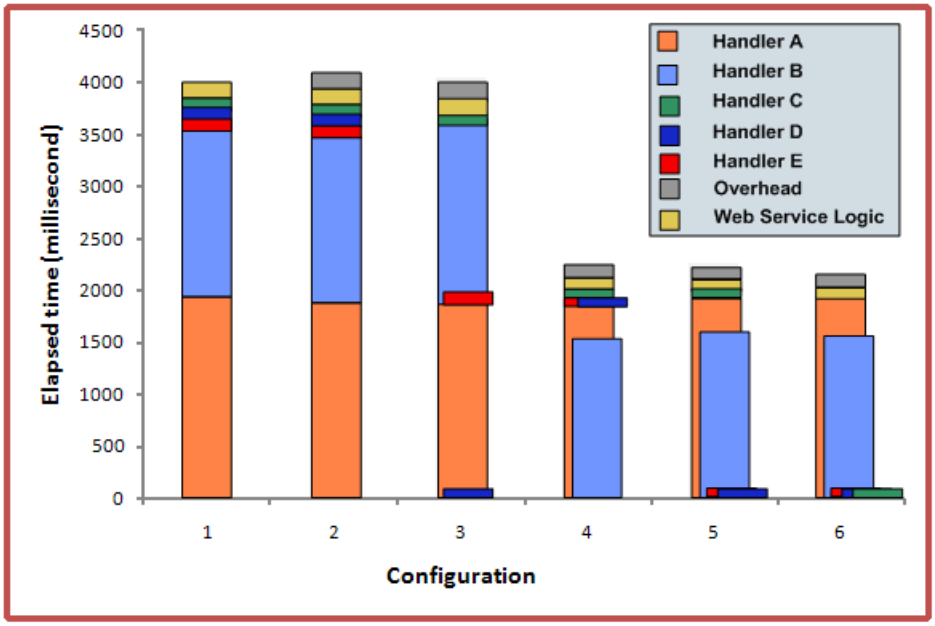


Figure 6-9: The service execution times of the six handler configurations containing the five handlers in the multiprocessor system

Table 6-12 shows the numerical values of the results for the multiprocessor system. Figure 6-10 depicts the standard deviations of the execution times for the multiprocessor system. The deviations are little higher than those in the multi-core

system. This can happen either due to the system scheduling algorithm or because of the system load during the execution time.

Table 6-12: The elapsed time for the service execution and the standard deviation of the performance benchmark in the multiprocessor system

Configuration number	1	2	3	4	5	6
Mean value (msec)	4023.02	4052.07	4025.95	2261.08	2250.96	2171.53
Standard Deviation	83.49	90.52	92.56	86.66	97.11	86.22

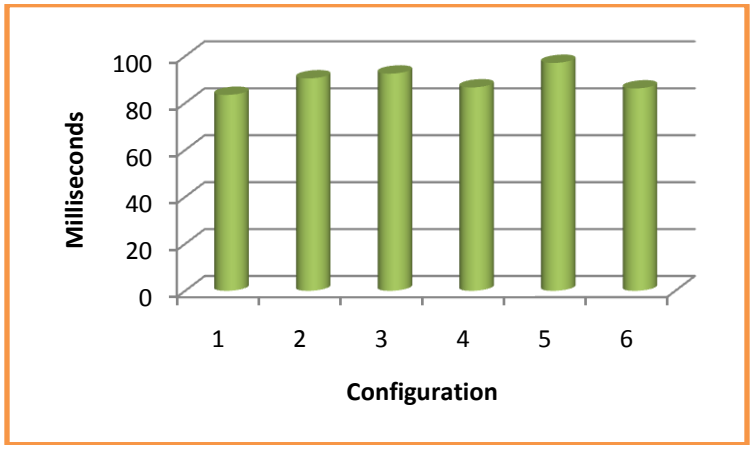


Figure 6-10 : Standard deviations of the service execution times in the multiprocessor system

Figure 6-11 illustrates results from the executions of the handlers in a cluster that communicates with a Local Area Network. The features of the computers have been provided in section 6.1.2. The execution times get smaller due to faster computers. However, this does not change the behavior of the handler configurations. They follow the same patterns of the previous systems. The sequential execution of DHArch is executed slower than those from the remaining configurations. The numerical values of the results are shown in Table 6-13.

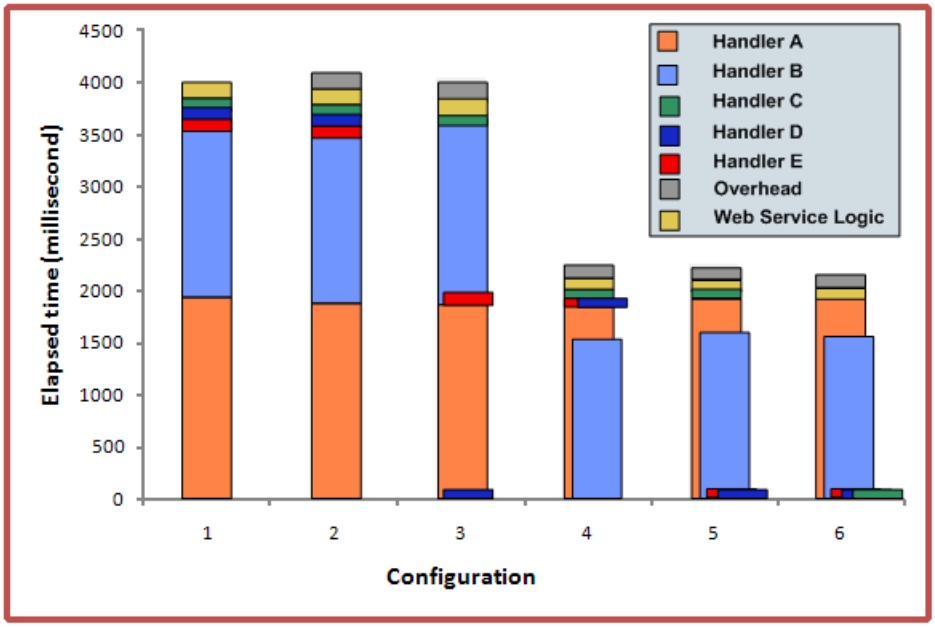


Figure 6-11 : The service execution times of the six handler configurations containing the five handlers in the cluster utilizing Local Area Network

The standard deviations, shown in Figure 6-12 are reasonable even if the tasks between handlers travel over the local network. The network is fast and consistent. The message transportation does not take too much time. When the results are compared with those from the previous systems, any side effect coming from the usage of LAN is not observed.

Table 6-13: The elapsed time for the service execution and the standard deviation of the performance benchmark in the cluster utilizing Local Area Network

Configuration number	1	2	3	4	5	6
Mean value (msec)	1717.08	1741.95	1712.22	1182.06	1150.55	1139.26
Standard Deviation (msec)	42.56	35.32	36.30	44.06	37.79	45.90

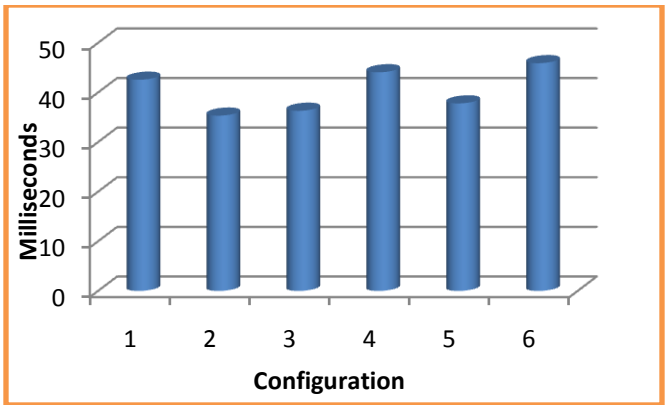


Figure 6-12: Standard deviations of the service execution times in the cluster utilizing Local Area Network

The results, shown in Figure 6-13 and Table 6-14, are from the single processor system, explained in section 6.1.2. In contrast to previous measurements, single processor system provides a different pattern. Thread scheduling becomes an issue. Since two handlers are heavily CPU-bound, the individual execution times of them are increasing when they are executed concurrently. Moreover, NaradaBrokering and Apache Axis in Apache Tomcat container use the same processor. This worsens the thread scheduling.

Table 6-14: The elapsed time for the service execution and the standard deviation of the performance benchmark in the single processor system

Configuration number	1	2	3	4	5	6
Mean value (msec)	1538.14	1661.73	1638.54	1558.9	1528.21	1488.67
Standard Deviation (msec)	56.32	58.29	54.86	73.82	85.90	86.80

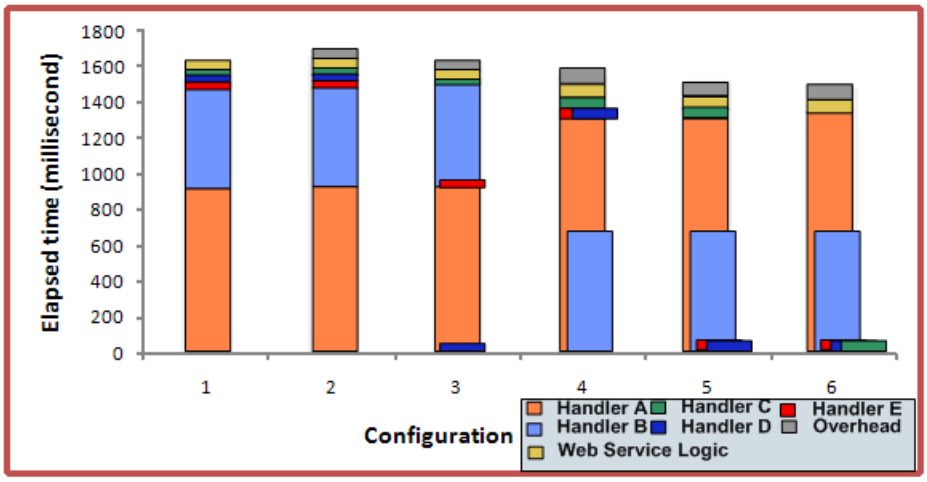


Figure 6-13: The service execution times of the six handler configurations containing the five handlers in the single processor system

Figure 6-14 illustrates the standard deviations of the execution times from the single processor system. Depending of the system load, the results fluctuate more than those from the previous systems. However, they, ranging from 50 to 80, are reasonable where the execution times are more than 1500 milliseconds.

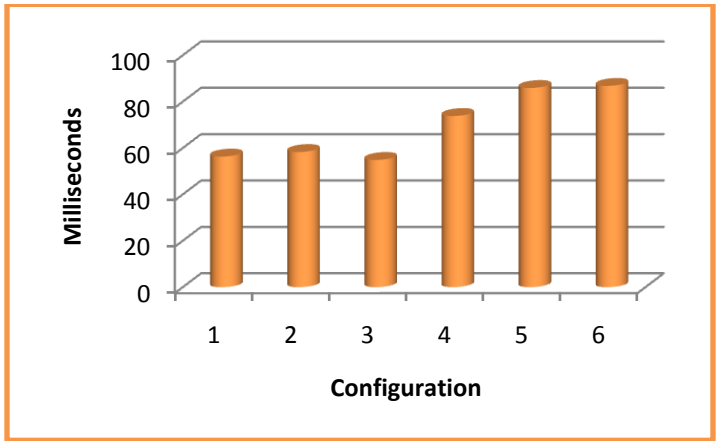


Figure 6-14: Standard deviations of the service execution times in the single processor system

6.1.5 Summary

We perform experiments to benchmark the different handler configurations in various environments. The handler configurations are built for the comparison purposes. Although handler distribution introduces an overhead, we also observed promising gains when they are executed concurrently.

6.2 Overhead of distributing a handler

Even though the distribution of handlers provides many advantages to Web Services, it is not for free. Pulling a handler out from a place and positioning it away from the place where Web Service endpoint is hosted brings additional costs. Relocation necessitates the transportation and the distribution management. This additional cost is inevitable. However, it can be kept in a reasonable range so that the relocation can be justified. In the remaining of this section, we will investigate the overhead for the dingle handler distribution in various environments.

6.2.1 Methodology

In order to calculate the overhead resulting from the handler distribution, we collect results for Apache Axis and DHArch. Basically, the distribution cost is the difference between the execution times of Apache Axis and DHArch. Apache Axis utilizes only sequential execution for the handlers in a single machine. On the other hand, DHArch is able to distribute these handlers into different computers. Hence, handler executions of these mechanisms utilizing the same environments with equal parameters contribute to calculate the overhead of a single handler distribution.

For the sake of the fairness, the results have been gathered by utilizing the same environments with equal parameters. The only difference is the distribution. Other than this difference, the remaining measurement parameters carefully selected equal. The utilized handlers for this benchmarking are completely same. The number of the handler in an execution is also equal in every step. Measurement starts from 1 handler. The number of the handler is increased by 10 in every step. We continue to add the same handler into the execution path until having 50 handlers. Figure 6-15 illustrates how the handlers are deployed in Apache Axis.

Figure 6-15: Apache Axis sequential handler deployment to measure the overhead

The same deployment strategy is applied in DHArch. However, there is only one important difference; handlers are distributed into different computers. Figure 6-16 illustrates the DHArch sequential deployment for the same number of handlers.

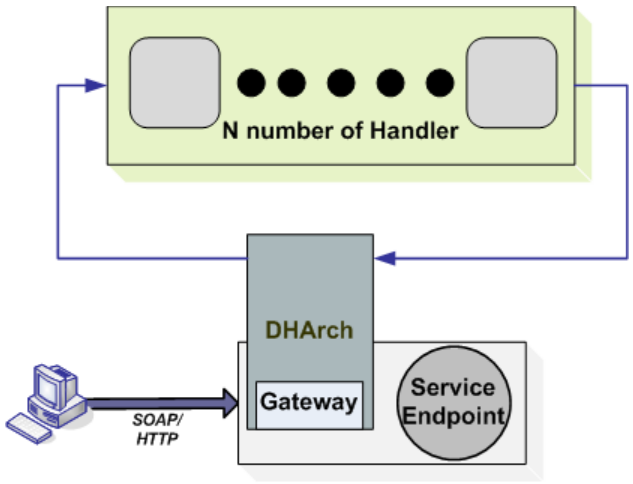


Figure 6-16 : DHArch sequential handler deployment to measure the overhead

Every measurement is observed 100 times; a client performs the same requests 100 times in every step. At the end of the measurement, the service elapsed times are collected and an average value from them is calculated. After gathering the values from both environments, the overhead is calculated with the following formula:

$$Overhead = (T_{dharch} - T_{axis}) / N \qquad \textbf{Equation I}$$

Where, T_{dharch} is the elapsed time of a service utilizing DHArch. T_{axis} is the elapsed time of a service utilizing Apache Axis. N is the number of the handlers in the deployment.

6.2.2 Environments

We have also performed the overhead measurements by using four different hardware configurations, which are explained in section 6.1.2. The first one is a multi-core system that utilizes 8 cores. The second system is a multiprocessor machine with 8 processors. The third environment contains 8 computers connected with a LAN. Finally, the fourth system is a single machine that utilizes a Redhat Linux operating system. The intention of using four different hardware configurations is to figure out how the distribution cost varies according to the hardware parameters.

6.2.3 Measurements

The first experiment is conducted in the multi-core system. We initially collected the execution time in Apache Axis handler structure. Then, the same scenario is repeated in DHArch. Figure 6-17 illustrates the execution time and standard deviation for Apache Axis and DHArch. The numerical values are provided in Table 6-15 and Table 6-16.

A performance improvement mechanism is not exploited in DHArch to find out the pure overhead added to the Apache Axis sequential execution; handlers are running sequentially with the same conditions in DHArch. Because of the overhead, DHArch service execution times are higher than those in Apache Axis. However, adding new handler into execution path increases the processing time linearly.

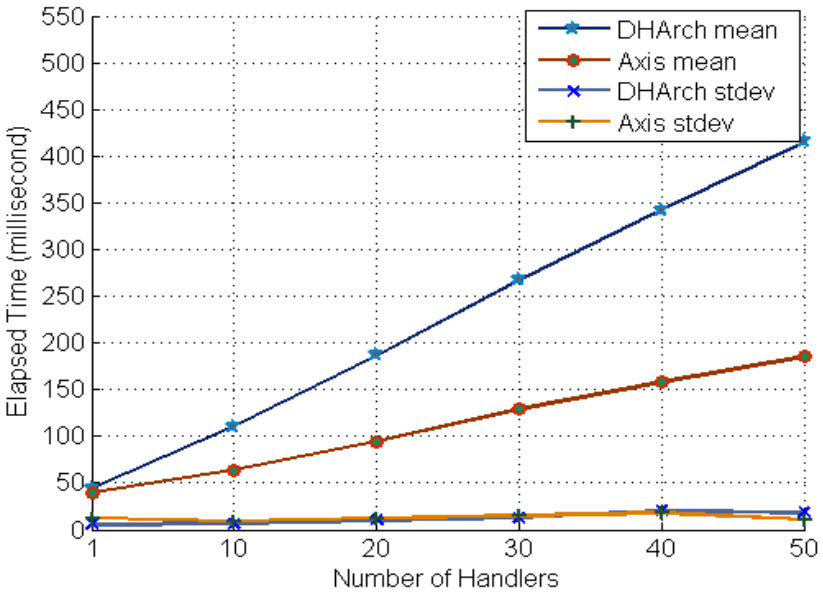


Figure 6-17: Comparison for the handler addition in Axis 1.x and DHArch for a multi-core system

When the formula in page 137 is applied to calculate the overheads, we observe that they are almost equal for the different number of handlers in the execution path. Table 6-17 shows the numerical values. The increasing number of handlers does not cause an unreasonable fluctuation. This is an expected outcome from a stable and scalable system.

Table 6-15 : DHArch execution results (in milliseconds) for the multi-core system while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	43.86	110.19	186.99	267.21	341.84	415.13
Standard deviation (msec)	5.61	6.45	10.36	13.33	20.22	18.46

Table 6-16 : Apache Axis execution results (in milliseconds) for the multi-core system while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	39.34	64.27	94.24	128.77	157.74	185.48
Standard deviation (msec)	12.74	8.99	12.84	14.90	17.88	10.77

Table 6-17 : Overheads of a handler distribution in the multi-core system for the increasing number of handlers in the execution path

Number of handlers	1	10	20	30	40	50
Overhead (msec)	4.52	4.59	4.63	4.61	4.60	4.59

The mean value of the overhead is provided in Table 6-18. The standard deviation is small since the fluctuation of the overheads for different number of handlers in the execution path is modest.

Table 6-18 : Mean value of the overheads for the multi-core system

Mean value (msec)	4.59
Standard deviation (msec)	0.039

The second deployment environment is the multiprocessor system. Figure 6-18 portrays the results for this hardware configuration. Similar to the multi-core system, adding new handlers causes a linear increase in the service execution times. Likewise, an elapsed execution time in DHArch is higher than the execution time in Apache Axis because of the overhead. The numerical values of the execution times are provided in Table 6-19 and Table 6-20.

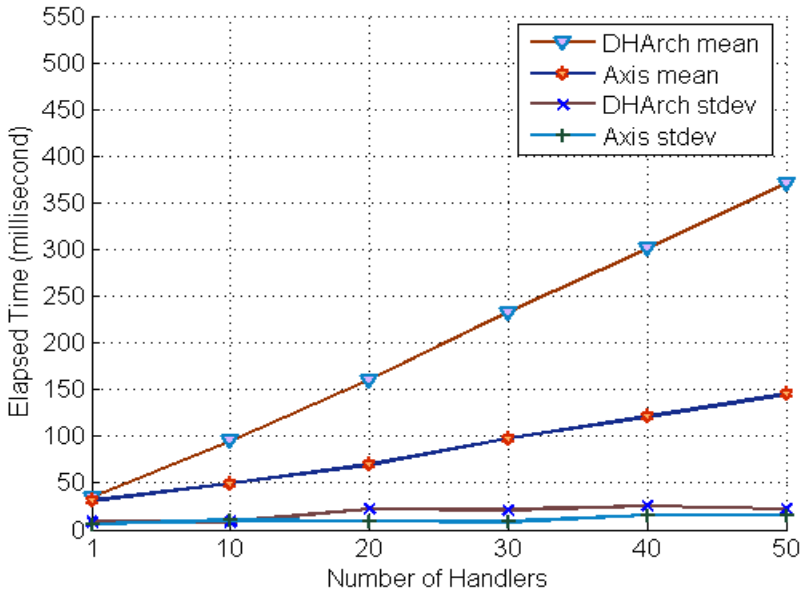


Figure 6-18 : Comparison for the handler addition in Axis 1.x and DHArch for a multiprocessor system

Table 6-19 : DHArch execution results (in milliseconds) for the multiprocessor system while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	35.2	95.26	160.37	232.74	301.07	370.74
Standard deviation (msec)	8.60	8.52	22.09	21.29	25.52	22.09

Table 6-20 : Apache Axis execution results (in milliseconds) for the multiprocessor system for the increasing number of handlers in the execution path

Number of handlers	1	10	20	30	40	50
Mean value (msec)	30.66	49.13	69.28	97.32	121.24	145.4
Standard deviation (msec)	6.19	10.41	9.61	8.23	15.92	15.28

Table 6-21 : Overheads of a handler distribution in the multiprocessor system for the increasing number of handlers in the execution path

Number of handlers	1	10	20	30	40	50
Overhead (msec)	4.54	4.61	4.55	4.51	4.49	4.50

The mean values of the overheads are provided in Table 6-22. The overhead in the multiprocessor system is smaller than that in the multi-core system. It is because of hardware effect; the faster computer reduces the cost.

Table 6-22 : Mean value of the overhead for the multiprocessor system

Mean value (msec)	4.53
Standard deviation (msec)	0.042

In contrast to the previous two hardware configurations, multiple computers sharing LAN have been utilized to collect the results in the third overhead measurement. Three computers are used for the deployment in DHArch. Service endpoint and the messaging broker are located into the individual computers. Similarly, handlers are distributed to another computer in the same network. The measurements gathered for Apache Axis and DHArch resemble to those in the previous two hardware configurations. Figure 6-19 depicts the results. The numerical values of the results are given in Table 6-23 and Table 6-24. Even though the tasks are carried between the distributed handlers by using LAN, the overhead is lower than those in the previous configurations. This illustrates that the processor speed affects the overhead more than the network speed. Previous configurations do not have network latency. However, we must know that the network speed in this hardware configuration has a minuscule effect due to the usage of the computers sharing a LAN; the distance between the computers is small.

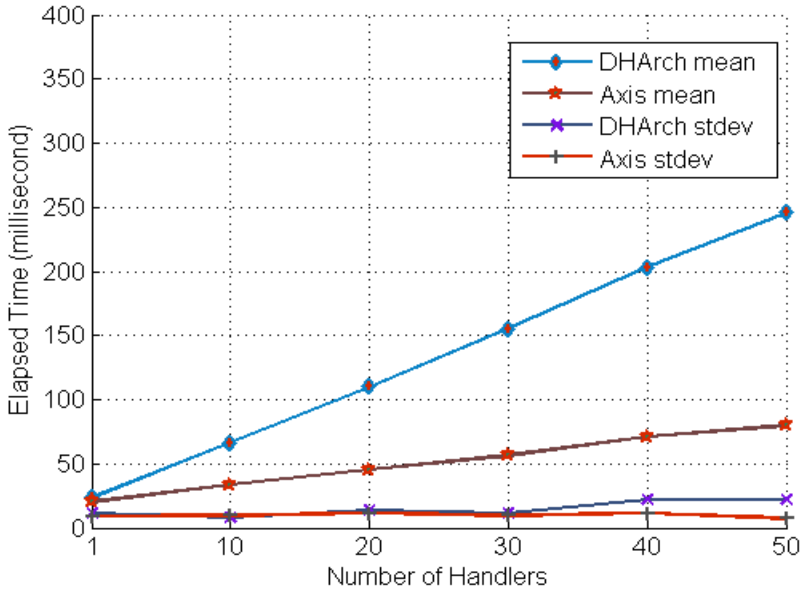


Figure 6-19 : Comparison for the handler addition in Axis 1.x and DHArch for a cluster using LAN

Table 6-23 : DHArch execution results (in milliseconds) for the system utilizing Local Area Network while the number of handlers is increasing.

Number of handlers	1	10	20	30	40	50
Mean value (msec)	23.54	66.51	110.28	155.52	203.16	245.87
Standard deviation (msec)	11.66	7.76	13.73	11.40	21.71	22.15

Table 6-24 : Apache Axis execution results (in milliseconds) for the system utilizing Local Area Network while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	20.24	33.4	45.16	56.56	70.92	80.08
Standard deviation (msec)	8.81	9.84	11.56	9.59	11.59	7.23

Table 6-25 : Overheads of a handler distribution in a cluster utilizing Local Area Network for the increasing number of handlers in the execution path

Number of handlers	1	10	20	30	40	50
Overhead (msec)	3.3	3.31	3.25	3.29	3.30	3.31

Table 6-25 provides the overhead values. The mean value of these is given in Table 6-26. Similar to previous configurations, the standard deviation is small because of the size of fluctuation in the overheads.

Table 6-26 : Mean value of the overheads in a cluster utilizing Local Area Network

Mean value (msec)	3.29
Standard deviation (msec)	0.21

Finally, we have conducted an experiment in the single processor system, explained in section 6.1.2. Figure 6-20 shows the results gathered in this configuration. The numerical values of these results are given in Table 6-27 Table 6-28. Similar to the previous hardware configurations, the execution times of a service in DHArch is higher those in Apache Axis because of the overhead resulting from the handler distribution. Although this system has a faster processor than the previous configurations and there is not a message transferring cost coming from the network usage, the overhead is not the smallest one. This must be due to thread scheduling. In this configuration, handlers are distributed into virtual machines instead of cores, processors or individual computers. In other words, handlers, endpoint and broker share a single processor. Hence, the thread scheduling causes performance degradation.

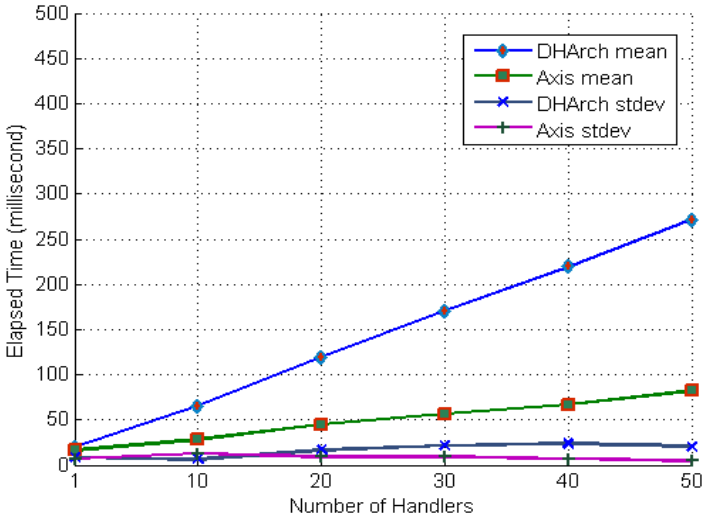


Figure 6-20 : Comparison for the handler addition in Axis 1.x and DHArch for a single processor system

Table 6-27 : DHArch execution results (in milliseconds) for a single processor system while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	20.42	65.19	119.38	170.72	219.55	271.55
Standard deviation (msec)	8.46	6.40	16.77	21.74	23.84	20.75

Table 6-28 : Apache Axis execution results (in milliseconds) for a single processor system while the number of handlers is increasing

Number of handlers	1	10	20	30	40	50
Mean value (msec)	16.68	27.84	44.95	56.56	66.99	81.98
Standard deviation (msec)	6.70	12.52	9.36	8.77	7.02	5.01

Table 6-29 Overheads of a handler distribution for a single processor system for the increasing number of handlers in the execution path

Number of handlers	1	10	20	30	40	50
Overhead (msec)	3.74	3.73	3.72	3.80	3.81	3.79

The mean value of the overhead and standard deviation is provided in Table 6-30.

Table 6-30 : Mean value of the overheads for the single processor system

Mean value (msec)	3.76
Standard deviation (msec)	0.04

6.2.4 Summary

We have conducted comprehensive experiments in four different hardware configurations to measure the overhead of a single handler distribution. The experiments have provided us a clear understanding of the behavior of DHArch in these diversified environments.

The average overhead ranges between 3-5 milliseconds. The overhead value is affected by the computing resources and the network connections. We utilize a handler that is not heavily CPU-bound. Handlers are generally contains both I/O and CPU-bound tasks. Thus, it is an appropriate handler to derive a general conclusion. We observed that computing power is one of the most important players on the overhead. Multiprocessor system has more powerful computing resources than the multi-core system. Hence, the overhead becomes smaller. Similarly, the computers sharing LAN are more powerful than those in the both multi-core and multiprocessor systems. Therefore, the overhead in LAN environment is better than those in the previous hardware configurations. This happens even though the messages in LAN has to be transferred to the distribute handlers hosted by an individual computer. The effect of the message transferring on the overhead is not much because the distance is small and LAN network provides fast message transferring environment. However, the network latency should be expected to become main factor for the overhead if the distance increases and network speed becomes slower.

6.3 Scalability

In this experiment, we will investigate the throughput in DHArch comparing with a conventional handler mechanism. We will also find answers for following questions: What is the effect of request rate over the processing time? How many handlers can be supported in DHArch?

6.3.1 Message rate

Web Services offer opportunities to perform tasks remotely. It is basically a paradigm that clients make requests to execute a task in a remote application. This structure may lead the situation that many clients make requests in a short time. For instance, an online shopping center which utilizes Web Service technologies may receive hundreds of transactions. There might be scenarios that the request rate may be even higher. For example, Web Service, which presents an interface to illustrate a real time tornado development, may receive inputs from thousands of sensors.

Consequently, a Web Service may have a very high request rate. Therefore, the system architecture must be efficient and effective that it can answer the increasing number of requests. Handler chain is one of the most crucial parts of the service execution. Its performance directly affects overall system performance. In the remainder of this section, we will investigate the scalability of DHArch by comparing with Apache Axis 1.x handler execution mechanism.

6.3.1.1 Environment and methodology

We utilize a multi-core machine cluster for benchmarking purpose. In this cluster, 8 computers communicate via a Local Area Network. Every computer has 2 Quad-core

Intel Xeon processors running at 2.33 GHz with 8 GB of memory and operating Red Hat Enterprise Linux ES release 4 (Nahant Update 4).

Three handlers are utilized for this measurement, Logger, Monitor and Format Converter. Logger stores the incoming messages into a file. Monitor keeps the information for the services such as the incoming message rate, the message size, and information about the clients, and number of clients which are connected and so on. The last handler, Format Converter, converts incoming message formats into a uniform format, the service format.

In a conventional handler deployment, only sequential handler execution is exploited. An output from one handler becomes an input for the next one. The order is determined according to their dependencies. If there is not any dependency, the order is not necessarily important. However, they have to be processed one after another; it is not possible to apply parallelism to handler executions even if there is not any dependency between them.

Apache Axis handler structure utilizes a pipeline of handlers that passes the massage from one handler to another. A configuration file, WSDD, defines the handlers and their position in the execution path. Apache Axis handler executions can be depicted as it is in Figure 6-21.

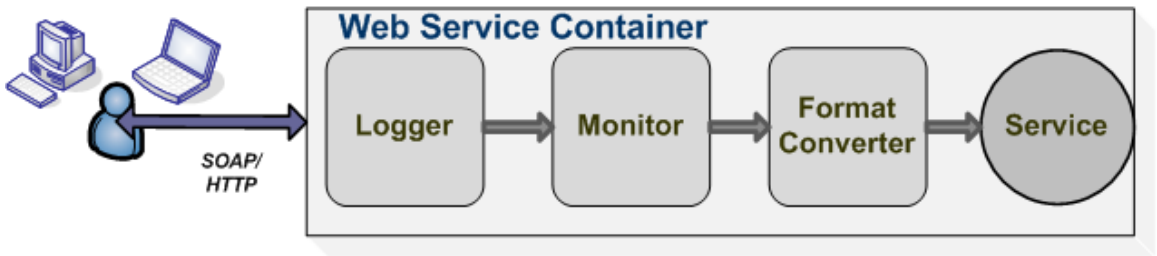


Figure 6-21 : Apache Axis sequential Handler deployment for scalability benchmarking

DHArch provides concurrent as well as sequential execution for the handlers. The group of handlers, used in Apache Axis benchmarking, is also utilized for DHArch. Even though handlers may not be possibly processed in a parallel manner because of dependencies, the handlers selected for this experiment are suitable for parallel execution. In other words, one handler's output is not necessarily important to next one. Hence parallel as well as sequential execution has been utilized for DHArch benchmarking. The deployments are portrayed in Figure 6-22 and Figure 6-23.

The sequence of handlers and handler parallelism is decided by DHArch orchestration module. Handlers are distributed to the individual virtual/physical machines for the sequential and parallel execution. For the sequential execution, DHArch sends the incoming messages to the handlers in the order of Apache Axis handler setup. In the case of parallel executions, messages are delivered to all handlers concurrently.

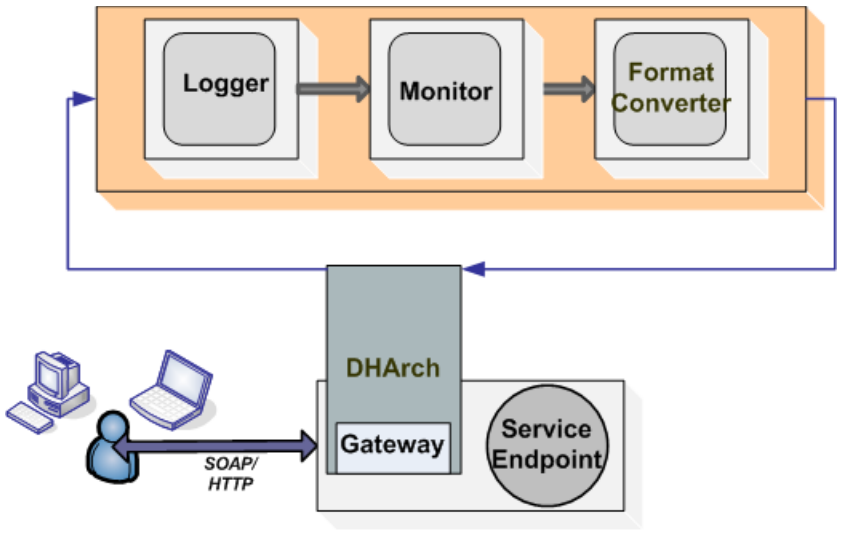


Figure 6-22 : DHArch sequential handler deployment for the scalability measurement

The results are gathered in a single as well as multiple computers. Apache Axis cannot benefit from the utilization of additional computers. Hence, their results are only

collected in a single computer. On the other hand, DHArch can perform handler execution in both environments.

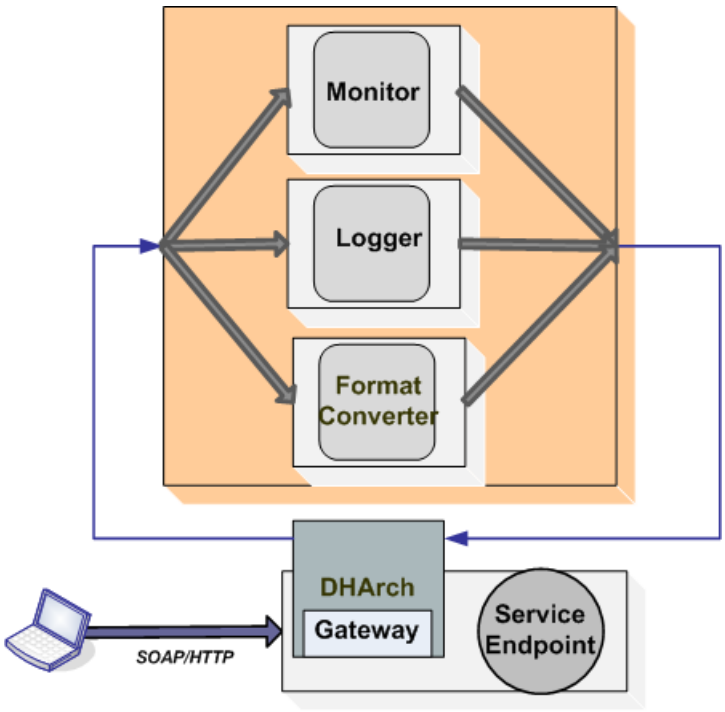


Figure 6-23: DHArch parallel handler deployment for the scalability measurement

Two experiments have been conducted. In the first experiment, we have measured the elapsed execution time of a single message while the number of messages per second is increasing. Clients start sending 1 message in a second within the duration of 100 seconds. In every step, the message rate was increased by 10 messages.

The second experiment has been performed to measure the cumulative time for the completion of the certain amount of messages. For this purpose, the messages were sent in a rate that the system computing resources has been fully utilized. We collected the cumulative number of executed messages in every second until all the message processing were accomplished.

6.3.1.2 Measurements

In order to measure the execution time for a single message while the number of message per second is increasing, we use the following experimental setup. The messages are sent within the same rate during 100 seconds. The rate starts from 1 message per second and continually increases 10 messages in every step to the level that the service can support. Figure 6-24 shows the results gathered from the single computer. It provides the elapsed execution times, measured in client side.

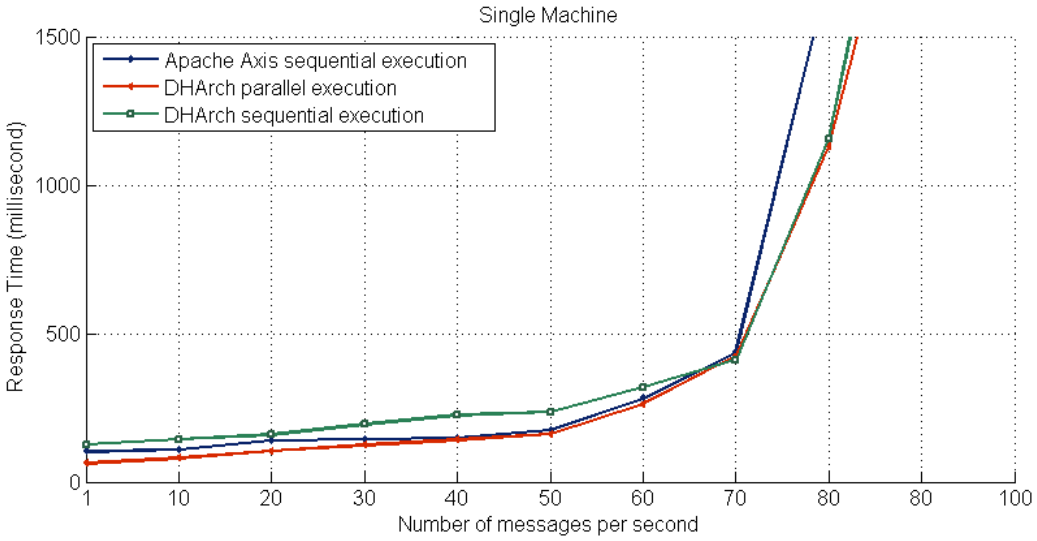


Figure 6-24: Message execution times for increasing message rate in a single machine

Until the system resources are saturated, the graph shows a pattern that we can deduce from the performance benchmarking results, see section 6.1. DHArch parallel execution has the fastest execution time while the sequential execution yields the highest processing time because of the distribution. Between these two, we see Apache Axis results. At one point, the processing times increases noticeably. This incident happens where the system resources are fully utilized. The message execution time has been

slowly increasing because every additional message starts sharing the computing resources. However, it has not been causing abrupt changes until the resources are fully used. When the resources start unable to meet the demands, the execution times has been skyrocketing. In Apache Axis, every arriving message starts another handler pipelining which shares the scarce resources. The context switches starts occuring more frequently. Hence, the execution time increases faster. There is not a regulation for the incoming messages to prevent this dilemma. On the other hand, DHArch has a different reason for the spike. DHArch does not allow the context switching cost worsening the system performance. Instead, the increase in execution time comes from the message waits in Incoming Message Queue (IMQueue). DHArch forces the messages wait in IMQueue; it keeps optimum number of messages in Executing Message Queue not to worsen the processing time because of the context switching. Hence, even though the pipelining provides very close results for a single computer when the system resources saturates, DHArch utilizes the system resources more effectively. Hence, we observe a slower increment in the execution time for the message rate between 70 and 80.

For the Apache Axis deployment, we observe that the message execution time started to decline significantly when the number of threads hits a point that thread scheduling becomes an issue. The performance begins deteriorating dramatically. The problem is that there are too many threads running and handler mechanism did not have any regulation to keep the performance in its optimum level. We notice that the fluctuation in the message processing increases considerably. When the engine completes enough message executions, the performance is improving and the system starts processing more messages. At the same time, the newly arriving messages begin building

up the new threads. When it reached its limits, the performance starts declining again. This pattern repeats itself until the message executions are completed. Table 6-31 depicts this phenomenon. The standard deviation for 80 messages per second illustrates the incident.

Table 6-31 : Apache Axis sequential execution results in single machine

Number of messages per second	The mean value of execution times of a message (millisecond)	Standard deviation
1	101.57	13.14
10	109.37	22.2
20	138.41	31.49
30	143.95	29.6
40	147.77	25.97
50	173.34	41.5
60	282.31	70.06
70	434.25	270.33
80	1745.65	909.56

On the other hand, since context switching does not affect the execution as it is in Apache Axis, the same fluctuation is not observed in DHArch. However, the increase in the execution time is not preventable when the system resources are drenched. In order to optimize message execution, the remaining messages that system cannot support are forced to wait in the queue. Hence, the message processing time increases steadily in DHArch. Table 6-32 illustrates this incident.

Table 6-32 : DHArch sequential execution results in a single machine

Number of messages per second	The mean value of execution times of a message (millisecond)	Standard deviation
1	125.09	20.63
10	143.7	36.82
20	158.38	28.93
30	193.76	39.76
40	223.72	44.48
50	236.21	78.98

60	319.05	55.89
70	410.2	104.74
80	1153.71	201.98
90	2618.14	302.32

A multi-core system provides advantages due to the individual core use for handler executions. If the resources are enough for the handlers which are running in a parallel manner, the computing resources do not have to be relinquished while the execution continues. For a single request, we definitely see the advantage of utilizing individual cores when the handler parallelism is applied. On the other hand, the advantage of the parallel execution of the handlers fades away for higher message rates. In other words, message parallelism, pipelining, becomes dominant factor in the executions. Both Apache Axis and DHArch benefits from pipelining. Hence, in this experiment, we investigate mainly pipelining rather than handler parallelism.

Table 6-33 : DHArch parallel execution results in a single machine

Number of messages per second	The mean value of execution times of a message (millisecond)	Standard deviation
1	61.65	11.77
10	80.64	11.4
20	104.93	19.07
30	123.98	24.62
40	141.11	24.29
50	161.76	40.73
60	262.44	98.35
70	424.47	61.39
80	1127.35	213.11
90	2340.43	353.45

When we introduce multiple computers, we see the immense gain in DHArch. Apache Axis cannot benefit from multiple computers but DHArch can. Hence, the

processing time stays stable longer time. Figure 6-25 portrays this situation. The message rate does not change the response time until 160 messages per second. One of the important events in the graph is the convergence of the Apache Axis single machine execution to the DHArch multiple machine sequential execution. In a single machine, Apache Axis processes massages faster than DHArch sequential execution. When we introduce the additional computers for DHArch, Apache Axis catches and later passes the execution time of DHArch sequential earlier.

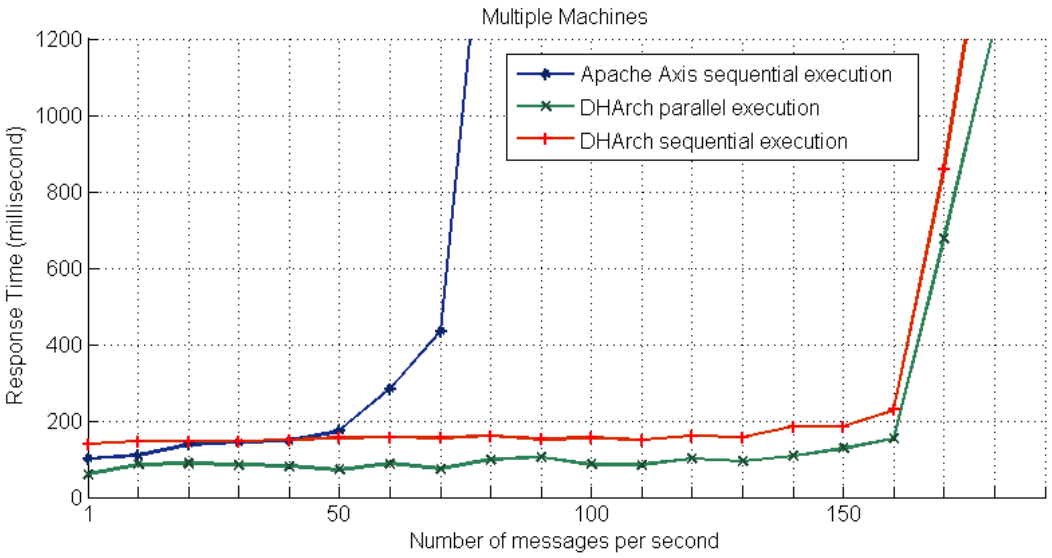


Figure 6-25 : Message execution times for increasing number of messages per second in multiple machines communicating via Local Area Network

Table 6-34 and Table 6-35 show the message execution times and standard deviations for multiple machines. Similar to single machine benchmark results, the response time of the service in sequential deployment is higher than the response time of parallel execution.

Table 6-34 : DHArch sequential execution results in multiple machines utilizing LAN

Number of messages per second	The mean value of execution times of a message (millisecond)	Standard deviation
1	138.78	11.26
10	147.23	19.98
20	146.23	25.11
30	147.25	37.33
40	149.43	21.86
50	154.97	25.09
60	156.52	25.43
70	155.53	17.64
80	160.81	25.76
90	151.52	24.68
100	155.7	41.53
110	150.11	29.55
120	160.6	85.34
130	156.91	22.84
140	184.95	37.08
150	184.95	63.17
160	228.95	80.28
170	857.72	112.8
180	1658.45	386.59

Table 6-35: DHArch parallel execution results in multiple machines utilizing LAN

Number of messages per second	The mean value of execution times of a message (millisecond)	Standard deviation
1	60.22	14.74
10	85.69	18.7
20	88.78	18.43
30	85.18	21.37
40	80.87	29.29

50	71.65	24.56
60	87.98	29.54
70	74.15	18.78
80	98.33	35.01
90	104.76	54.47
100	85.48	30.99
110	84.36	27.65
120	100.18	33.08
130	93.82	34.44
140	107.9	48.32
150	127.74	87.38
160	154.35	114.85
170	675.95	96.75
180	1230.91	116.19

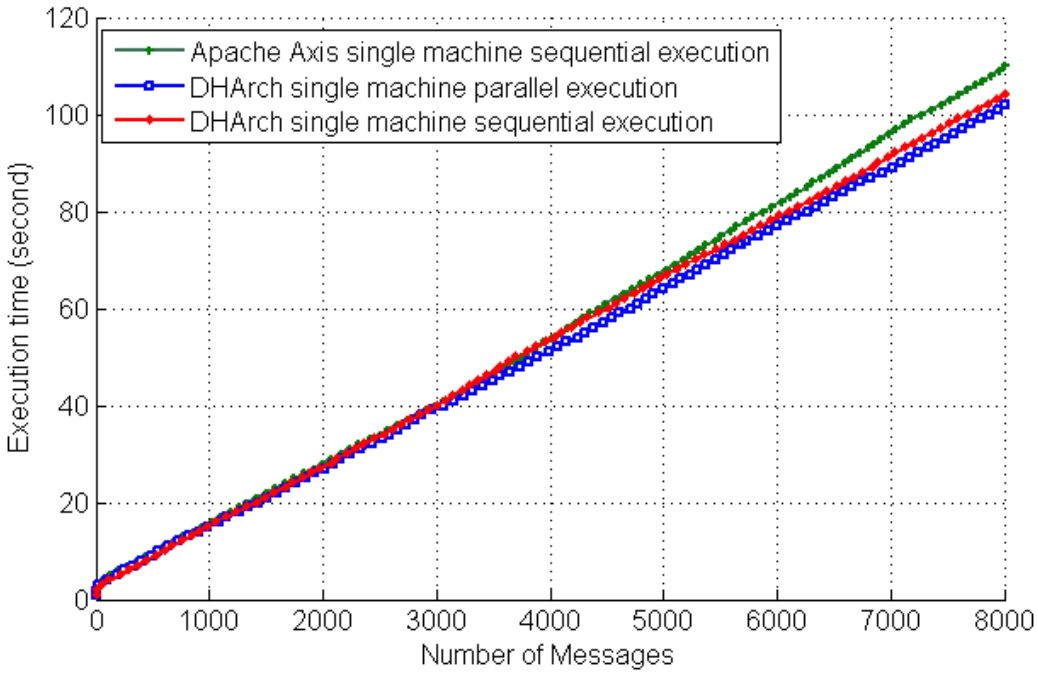


Figure 6-26 : Execution times for increasing number of messages in a single machine

In the second experiment, message rate is 80 messages per second where the system resources start being utilized fully in a single machine. The message rate is kept same for 100 seconds. In other words, 8000 messages are sent in total. In every second,

we measure the cumulative number of the executed messages. The results are depicted in Figure 6-26.

When we look at the graph, we notice that Apache Axis completes its executions later than DHArch. The reason is the thread scheduling. DHArch employs a regulatory mechanism to control thread scheduling. Queues regulate the flow control and keep the execution in optimum level. This does not prevent accepting the incoming messages. The arriving messages are kept in another queue, Incoming Message Queue. When a message is arrived, its execution does not start at once. It waits in the queue to be selected for the execution. The only messages being executed are those in the execution queue. It does not allow creating too many parallel message execution pipelines that shares the resources and causes performance degradation. Another observation from the figure is the closeness of the parallel and sequential executions of DHArch. While the system resources are being used fully, the parallel or sequential execution does not differ so much because the dominant factor is pipelining rather than handler parallelism.

When additional computers are introduced to DHArch, the performance becomes very promising. The processing time of the same amount of messages is reduced more than two fold and number of messages executed in a given time is increased considerably. Figure 6-27 portrays the results.

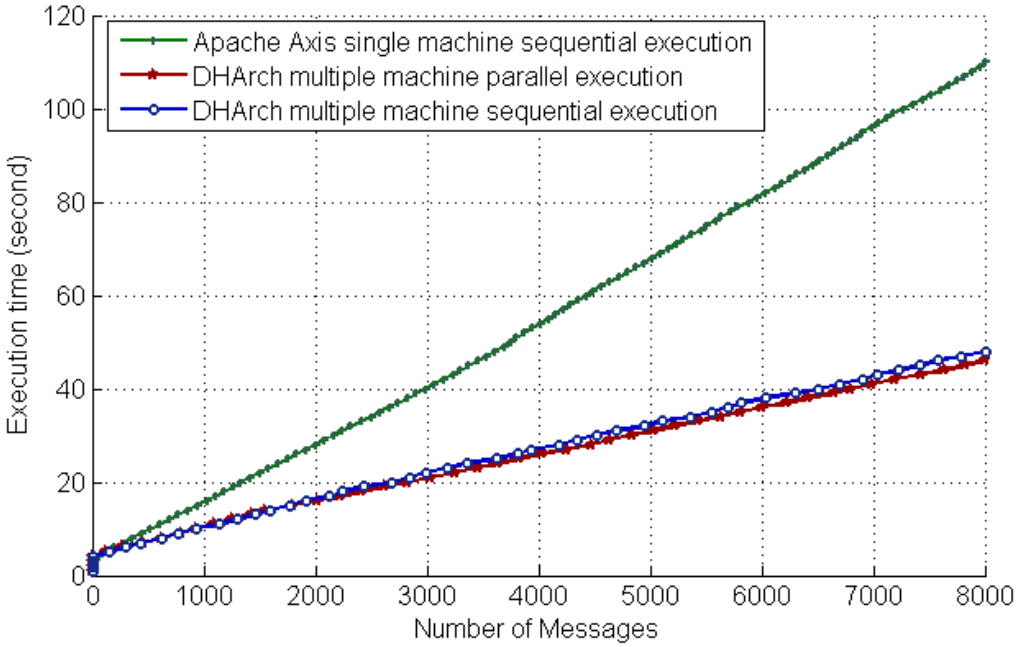


Figure 6-27 : Execution time for increasing number of messages in multiple machines

We clearly notice the advantages of utilizing DHArch in terms of throughput when multiple computes are used for the computation, shown in Table 6-36. In single machine, the message rate is 80 messages per second. The throughputs are very close to one another. When the multiple machines are used in DHArch, the throughput becomes favorable to the DHArch because the number of the processed messages doubles.

Table 6-36 : Throughput where the system resources are being utilized fully.

	Throughput (messages per second)
Apache Axis in a single machine	72
DHArch sequential in a single machine	78
DHArch parallel in a single machine	76
DHArch sequential in multiple machines	166
DHArch parallel in multiple machines	173

6.3.2 Scalability in the number of handlers

DHArch provides an efficient and effective environment for the handlers. We conducted experiments to find out the boundaries for the total number of handlers joining an execution. Although there is not, theoretically, an upper bound for the number of handlers that a Web Service can have, assuming over 20 handlers in a Web Service is an overestimation.

Handlers are distributed by utilizing non-blocking I/O TCP communication type of NaradaBrokering. There are several connection types in NaradaBrokering such as TCP, NIOTCP, and UDP and so on. As in every limited resource, there exist boundaries of the number of connections. We conduct experiments in different environments.

Multi-core and multiprocessor systems, explained in section 6.1.2 can support up to 300 distributed handlers. However, the cluster sharing LAN and the single processor system, explained in section 6.1.2 supports around 200 distributed handlers. The differences between these systems are resources and operating systems. First two systems utilize Solaris operating system while the remaining runs Redhat Linux operating system. The memory size is in favor of first two systems too.

In any case, we do not see any problem because the numbers are well over the expected number of handlers in a service. On the other hand, we have two options if we need more than 300 handlers in an execution. The number of connection can be increased by switching to TCP type communication that supports 1K connections. The second solution is the utilization of the broker network capability of NaradBrokering. The above boundaries are for a single broker. By introducing additional brokers, we are able to

remove the limitations. Broker network scales very well. Hence, the cost of adding a new broker to the system is very little.

6.3.3 Summary

DHArch scalability is measured in terms of message rate and the number of handlers that can be utilized in an execution. Message rate is very important because many Web Service applications receive many requests in a short amount of time. An improvement in handler structure would contribute overall because it is one of the main computing components of a Web Service execution.

Apache Axis employs a new thread to process arriving messages if there is an available one in the system. In other words, it tries to provide services to many messages at the same time. This parallel execution of messages contributes to the throughput of the system. However, the performance starts degrading when the number of the message reaches its limits. Thread scheduling diminishes the efficiency when the system resources are depleted and the context switching occurs more frequently.

Similarly, DHArch supports parallel message execution, pipelining. However, there is an improvement. Instead of letting every message arriving to the system starts its execution right away; DHArch processed optimum number of messages and keeps the remaining in Incoming Message Queue. This regulation prevents the performance degradation because of too many messages running concurrently. It keeps DHArch operating in its most efficient level.

Moreover, DHArch is able to utilize additional computers to remove the limitations over the scarce computing resources. This affects the throughput very dramatically. More requests are answered in certain duration. Finally, DHArch supports

much more handlers with a single broker than a Web Service execution requires. Additionally, it is capable of increasing the number of handlers by introducing a broker network when there is a need.

6.4 Deploying Web Services Resource Framework and Web Services Eventing

We want to crown the experiments by showing deployment of two well-known Web Service Specifications. Many efforts have been dedicated to the WS-Specification. The implementations gradually have started to appear Web Service arena. We have found several groups providing the WS-Specification implementations. Among them, two specs were fitting to our purpose; WS-Resource Framework [21] and WS-Eventing[118].

6.4.1 Web Services Resource Framework (WSRF)

Web Services must offer ability to the clients to access and manipulate state. Even though managing states is challenging, stateful resources are not utterly evitable from the services. A service may utilize one or more stateful resources. Hence, Web Service Architecture should provide eligible functionalities to access them. On the other hand, while this capability is being offered, having a convention is essential. Web Service Resource Framework (WSRF) establishes the necessary convention for the states. It provides capabilities to insert, update, and discover the stateful resources in a standard and interoperable way.

We utilize the Apache implementation of WSRF for the experimental purpose. We created our stateful resource for *sensors*[1]. Following XML element is designed to request data for a sensor. Star sign allows requesting all the data. Single information can also be inquired.

```
<wsrp:QueryResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd">
  <wsrp:QueryExpression Dialect="http://www.w3.org/TR/1999/REC-xpath-
19991116">*</wsrp:QueryExpression>
</wsrp:QueryResourceProperties>
```

In addition to inquiry, insert and update functionalities can also be achieved in a standard way. The following XML elements show how to insert and update information for a sensor.

```
<wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                    xmlns:sn="http://ws.apache.org/resource/example/sensor">
  <wsrp:Insert>
    <sn:Comment>set via wsrp:Insert</sn:Comment>
  </wsrp:Insert>
</wsrp:SetResourceProperties>
```

```
<wsrp:Update>
    <sn:LastTimeOfSignal>10:20:32 AM February 23,2007</sn:LastTimeOfSignal>
</wsrp:Update>
```

6.4.2 Web Services Eventing (WS-Eventing)

A Web Service may benefit from receiving a notification when an event occurs. Instead of checking an event occurrence repeatedly, an entity can be notified by an event source when an event happens. In this paradigm, a service, called as subscriber, needs to register itself to a certain interest with another service, called as event source. Web

[1] WSDL file and the detailed SOAP messages for sensor satateful resource are provided in Appendix E

Service Eventing (WS-Eventing) defines a protocol to standardize this effort. A subscription manager can be employed to administer subscriptions. We utilize FIN, an implementation of WS-Eventing [119] from Pervasive Technology Lab. It provides handler based implementation as well as service based implementation[2].

The following XML element shows how a sink entity requests a subscription. The request is registered to the Subscription Manager. When a source publishes an event to the topic, */sensor/cal*, the sink is notified.

```
<even:Subscribe>
  <even:EndTo>
<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
  </even:EndTo>
   <even:Delivery mode=
"http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
  <even:Expires xsi:type="xs:dateTime" xmlns:xs=
"http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-04:00
</even:Expires>
  <even:Filter   Dialect=
"http://www.naradabrokering.org/TopicMatching">/sensor/cal</even:Filter>
  <even:NotifyTo>
    <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
  </even:NotifyTo>
</even:Subscribe>
```

An event is carried to the subscriber by an XML document. The following XML element notifies an important activity for a sensor, located California.

```
 <sens:sensor>
    <sens:cal>
      <sens:number>1</sens:number>
      <sens:CurrentTime>2007-03-01T00:41:14.856-05:00</sens:CurrentTime>
      <sens:Location>california</sens:Location>
      <nar:Application-Content>Tracker 1 : Important activity
happened</nar:Application-Content>
    </sens:cal>
 </sens:sensor>
```

[2] The SOAP messages of the WS-Eventing interactions are provided in Appendix F

6.4.3 Environment

A computer cluster is utilized for this experiment. It contains 8 machines having the same features. They share Local Area Network to communicate each other and utilize Fedora Core release 1 (Yarrow) in Intel Xeon CPU running on 2.40GHz and 2GB memory.

Before starting benchmarking, the initializations of the specifications are completed. Sink registers itself to the topic /sensor/cal and sensor satateful resource stores the initial information. Most importantly, the suitable massages are selected, one from WS-Eventing and one from WSRF. These messages are combined to create a new message in order to run WSRF and WS-Eventing handlers in a parallel manner. Several technical problems originating from the implementations of the specifications have been witnessed during this procedure, though. The following SOAP message is the created combined message for the experiment:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:wsewsrf="http://www.dharch.org/wsewsrf"
        xmlns:top="http://www.naradabrokering.org/TopicMatching"
        xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
        xmlns:sens="http://www.naradabrokering.org/sensor"
        xmlns:nar="http://www.naradabrokering.org"
        xmlns:sn="http://ws.apache.org/resource/example/sensor"
        xmlns:wsewsrf="http://ws.dharch.org/wsewsrf ">
  <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
         <wsewsrf:wsrf>
        <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
        <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/you
rWsdlRequestName</wsa:Action>
        <sn:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
      </wsewsrf:wsrf>
      <wsewsrf:wse>
```

```
        <top:Topic>/sensor/cal</top:Topic>
          <add:MessageID>c3e00553-db0c-4ae0-965a-a59183ed3761</add:MessageID>
          <add:From>
<add:Address>http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</add:Address>
        </add:From>
      </wsewsrf:wse>
  </Header>
  <Body>
    <wsewsrf:comb>
    <sens:sensor>
     <sens:cal>
      <sens:number>1</sens:number>
      <sens:CurrentTime>2007-03-01T00:41:14.856-05:00</sens:CurrentTime>
      <sens:Location>california</sens:Location>
      <nar:Application-Content>Tracker 1 : Important activity
happened</nar:Application-Content>
     </sens:cal>
    </sens:sensor>
    <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
          <wsrp:Update>
           <sn:Options>
           <sn:Option>Do we need to restart this?</sn:Option>
           <sn:Option>yes</sn:Option>
           <sn:Option>Do we need to keep previous month data?</sn:Option>
           <sn:Option>no</sn:Option>
           <sn:Option>Is it necessary to inform the people if abnormal activity is
observed?</sn:Option>
           <sn:Option>yes</sn:Option>
           </sn:Options>
          </wsrp:Update>
    </wsrp:SetResourceProperties>
    </wsewsrf:comb>
  </Body>
</Envelope>
```

The message above notifies an important activity and updates information for a sensor stateful resource. When it is received, WS-Eventing source handler looks for the subscription manager service to find out the interested subscribers. Then, it delivers the event to the sinks, the interested subscribers. While notification is happening, WSRF

handler also updates the values of the states, which are kept in storage, and forwards the information with the additional data previously stored.

6.4.3.1 Deploying the specifications for Apache Axis

Specifications are, first, deployed for Apache Axis. Every single request is observed 100 times. Handlers and service endpoint utilize a single computer. The remaining components of the specifications are hosted by the individual computers in the cluster. The logical deployment is depicted in Figure 6-28.

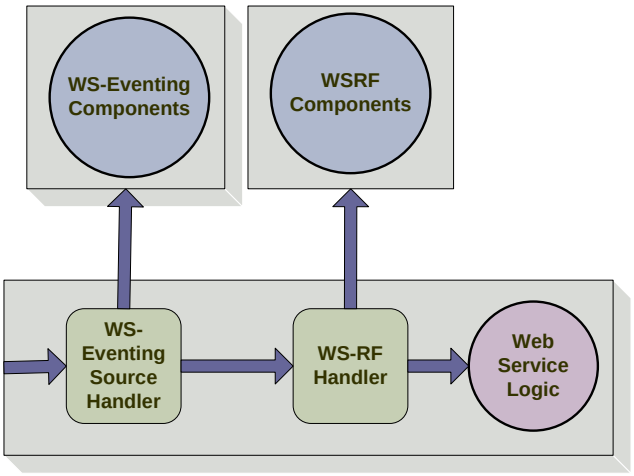


Figure 6-28: Sequential Execution of WSRF and WS-Eventing

6.4.3.2 Deploying the specifications for DHArch

The environment to execute WS- Eventing and WSRF is also created for DHArch.WS-Eventing requires individual computers for its components; Sink Source and Subscription Manager. Hence, they are located to the separate computers. Similarly, WSRF as well as NaradaBrokering are located into the individual computers in the cluster. Finally, the service endpoint is placed its location in the cluster. The deployment can be portrayed as in Figure 6-29.

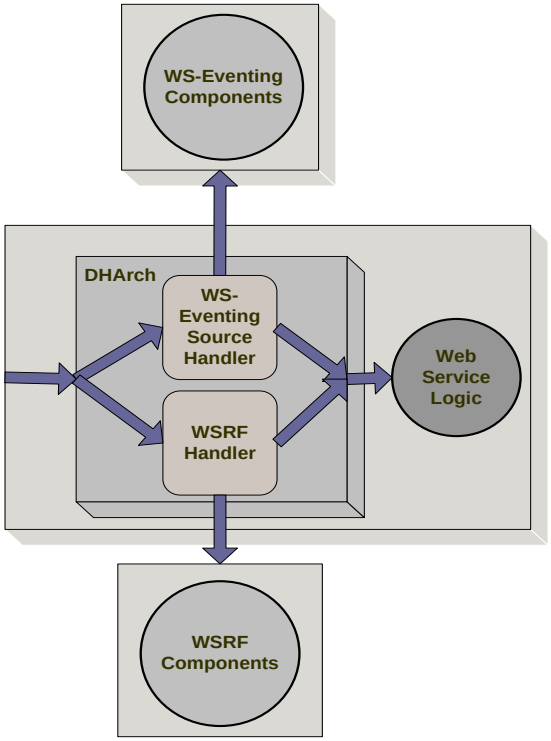


Figure 6-29 : Parallel Execution of WSRF and WS- Eventing

6.4.4 Results and analysis

First, we gathered the results in Apache Axis handler structure by running WS-Eventing and WSRF sequentially. The handlers are deployed into the request path. They look for their responsible elements in the messages. In other words, the handlers process only the relevant elements. We have individually measured the execution times of the WSRF and WS-Eventing. The results are shown in Table 6-37.

Table 6-37: WSRF and WS-Eventing sequential execution in Axis handler structure

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	69.32	55.08	162.14
Standard deviation	6.51	4.98	7.18

We perform the same sequential handler execution in DHArch. Because of the overhead originating from the distribution of the handlers, the time of processing a single message increases. The results are shown in Table 6-38.

Table 6-38: WSRF and WS-Eventing sequential execution in DHArch

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	70.25	54.68	171.64
Standard deviation	4.45	3.93	10.08

When we introduce the parallelism, we see significant improvement in the service performance. The concurrency reduces the execution cost of a single request by one forth. The cumulative execution time of the handlers in a sequential processing is around 124 milliseconds. It is slightly higher than the total execution time of the service in DHArch parallel handler execution. Since WSRF processing time is higher, it is the main player to determine the processing time of the handlers joining to the parallel execution. Due to the fact that DHArch deals with only handlers, the service endpoint processing time does not change. A service without handler executions takes almost 40 milliseconds. Table 6-39 shows the execution times and standard deviations of DHArch parallel handler execution.

Table 6-39 : WSRF and WS-Eventing parallel execution in DHArch

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	69.49	54.45	115.15
Standard deviation	5.53	3.42	12.15

Figure 6-30 shows the results from the execution of WSRF and WS-Eventing for Apache Axis and DHArch. The benchmarking demonstrates the advantage of parallelism for the handler execution. While the search goes on for the handler candidates among the specification, we encounter a very small domain of handlers which is possibly executable

concurrently. Even in this domain, the way of implementation causes problems for the distribution. We are expecting that this domain grows in near future. Hence, utilizing the distribution and parallelism for the specifications will produce many state-of-art applications.

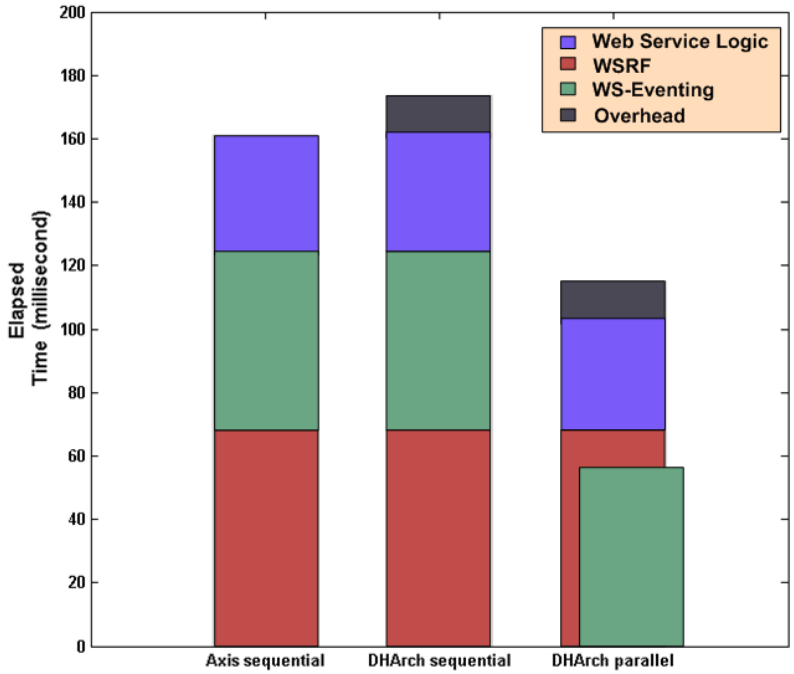


Figure 6-30 : Executing WSRF and WS-Eventing

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH ISSUES

7.1 Thesis summary

Service Oriented Architectures, specifically Web Service technologies, focus on benefiting maximally from interoperability and reusability. Many standards and structures have been developed to provide an interoperable environment. Web Service Description Language (WSDL), Universal Description Discovery and Integration (UDDI) and Simple Object Access protocol (SOAP) are de-facto standards to build Web Service. WSDL is a contract to agree on how to use a service. Agreeing on something is very widely accepted notion. USB devices agree on a communication interface with the computers. Similarly, electric devices contract to get the electricity by using a plug. UDDI provides registry for the services. It contributes to Web Services by listing them in

a publicly known place. Finally, SOAP is a message format allowing the communication between clients and services.

A Web Service is basically an application offering a service via SOAP messaging. On the top of this, many WS-Specifications have been introduced to provide additional capabilities. Many others are already on the way. Furthermore, there are efforts to build efficient Web Service processing environments. These environments compose many tools to process SOAP messages, which is the most basic and essential task of a service execution framework. Hence, SOAP processing engines, Web Service containers, have been constructed to provide an efficient environment and to hide the complexity of the SOAP processing from the user.

Web Services exploit additive functionalities to improve its capabilities such as security, reliability, logging and so on. Some of these functionalities have been standardized as WS-Specifications such as WS-Security and WS-Reliable Messaging. In many cases, the functionalities are very essential for a service. For example, a health service without reliability may be deadly. A monitoring service without logging may be useless.

Consequently, a Web Service needs additional functionalities to improve its capabilities. These additive functionalities are called handlers or filters. They are inevitable for many services as the necessary capabilities are stated for the health and monitoring services. This necessity forces the containers to create their internal handler architecture. However, the design is very critical in order to be successful in this effort. Since handlers are one of the key SOAP processing component of Web Service Architecture, this design affects the whole Web Service execution structure. Therefore,

we have investigated the handler architectures extensively and derived very vital and important results from this conclusive research. Distributed Handler Architecture (DHArch) shows us many essential features that are necessary for efficient, scalable, flexible, and modular handler architecture.

DHArch has many key features. It provides very efficient handler architecture by exploiting concurrent handler execution and utilizing additional resources. Many handlers are independent from each other. In other words, they can be processed concurrently without harming the correctness of the execution. This improves the performance dramatically. Moreover, the efficiency significantly increases when the parallel executions leverages additional resources. For example, taking advantage of an individual powerful machine for WS-Security in LAN network contributes to the system efficiency incredibly.

DHArch benefits from message parallelism in addition to the handler parallelism. Instead of waiting for the completion of a message execution, many messages can be processed at the same time. We called this *message pipelining*. DHArch utilizes pipelining by leveraging its internal structures. DHArch processes the optimum number of messages and keeps the remaining in a queue instead of letting every message arriving to the system to start its execution right away. This regulation prevents the performance degradation because of too many messages running concurrently. Additionally, NaradaBrokering also contributes to the flow control with its queuing capability. It keeps the messages for the handlers until a handler becomes available to accept them.

Orchestration is a significant feature to collaborate the distributed applications. Dissemination of the handlers requires a handler orchestration. Promising results cannot

be expected without a decent orchestration mechanism for the handlers. Hence, an orchestration mechanism has been introduced. It provides two main advantages. First of all, it offers very efficient and effective engine by introduction of the separation of the description and the execution while it is providing very powerful expressiveness. Secondly, this mechanism helps to build dynamic handler structure.

DHArch scales very well. Having additional resources improves the scalability. More resources allow answering more requests. Since a Web Service may contain many handlers in addition to the Service endpoint, they all together may saturate a single machine. It gets worse while many clients are requesting many services concurrently. The response time keeps increasing. Instead, the bottleneck points can be eliminated by introducing additional resources and utilization of the concurrency.

DHArch is a very flexible system. It easily allows adding new handlers. The architecture can also easily be adapted to a Web Service Container. Switching from Apache Axis 1.x to Apache Axis 2 requires minor changes. The only necessary action is the implanting a suitable gateway. Furthermore, it is also able to utilize a variety of platforms for the handler distribution. It can process handlers in a system ranging from a single computer, multi-core, and multi processor to many computers.

7.2 Answering the research questions

In this section, we will answer the questions raised in the first chapter.

Are the conventional handler architectures enough? How can we improve the architecture? Why do we need to improve it?

When we look at the conventional handler structures, we realized that there is a wall before us. Services are getting complicated by continually adding new capabilities.

Utilizing a single machine will not suffice. Moreover, some handler executions take too much time so that they cause bottlenecks. This issue has to be addressed.

Moore's law predicts that the processors will continually improve. The network is also getting faster in every day. Hence many resources become available to be utilized. Handlers can leverage these offerings by being distributed. Distribution provides more efficient, scalable and modular handler structures. Chapter 1 discussed these topics in detail.

What does handler distribution require?

Handler Distribution necessitates data structures, orchestration and messaging infrastructure. In Chapter 3, we discuss the necessary structures and mechanisms under the title of DHArch modules. Efficient messaging infrastructure and orchestration mechanism are very critical. Additionally, message context registry and the messaging format needs to be carefully designed. Moreover, control mechanisms are required in order to assure the necessary quality of the system. Flow control is one of them; DHArch utilizes queues to optimize the flow.

What is the role of messaging? How can this very key supporter of an interoperable system be utilized?

Messaging perfectly fits a task to transport the messages to the distributed handlers. It is the native aspect for Web Services. Messaging decouples the components and improves the interoperability. Although asynchronous messaging is hard to manage, it offers best capabilities for the distribution. Messaging and its usage are explored in section 3.2.2.

How can we provide efficient and effective handler orchestration?

Handler orchestration is investigated extensively in Chapter 4. Orchestration is introduced in a way that offers two important key features; simplicity and powerful expressiveness. The engine is kept so simple that it has no apparent deficiency. On the other hand, the description is very powerful. Hence, very complex structures can be described.

How does distributed handler execution happen?

Handlers are able to distribute by utilizing messaging and handler orchestration mechanism. Figure 5-3 provides the general picture for the distributed handler execution in DHArch. Many necessary actions and decision have to be taken. The detailed information about how a distributed execution happens can be found in Chapter 5.

Performance wise, is handler distribution plausible?

Parallel execution and utilization of additional resources boost the performance. We conduct comprehensive experiments and analyze the outputs in section 6.1. The results have been gathered in various platforms to have a general conclusion about the necessary requirements for having plausible results.

Is there any overhead for the distribution?

Since the distribution necessitates the transferring the messages between the computing entities, an overhead occurs. Moreover, the management of distributed processing causes an additional cost. Section 6.2 investigates a handler distribution overhead in detail. The main actors of the overhead are also discussed.

Does the handler distribution scale very well?

Leveraging additional resources and utilizing parallel processing contributes to the scalable handler processing architecture. DHArch scales very well in terms of both

the number of handlers and the number of messages being processed. Section 6.3 explores the scalability of the DHArch and derives the useful conclusions.

What are the criteria of distributing a handler? What are the architectural principles of the handler distribution?

When we come to the point of deciding whether we distribute a handler, there is a criterion performance wise. The overhead should be compensated by the gain so that the distribution becomes plausible. Additionally, not all handlers are suitable for the distribution. Because of their nature, some handlers are better to stay within the same environment of the Web Service endpoint. For example, the distribution of a reliability handler in an unreliable environment necessitates additional reliability. A security handler can be distributed in a secure LAN. However, WAN would not be appealing unless the additional security costs are in reasonable range.

7.3 Future research

A Web Service container is basically a Web Service operating environment offering capabilities to process SOAP messages. The capabilities can be classified into two categories. The first category contains the applications offering general abilities such as SOAP serialization/deserialization, transport related features, and so forth. This category contains built-in capabilities and they are out of scope of this dissertation. The second category contains the applications that are provided by users to support Web Services. In this research, our focus has been on this category to find out how we improve the design of this portion of the SOAP processing environment. On the other hand, in this effort, we shed light on the necessary principles in designing a distributed Web Service Operating System; a Distributed Web Service Container is a very appealing research area

so that the limitations and many obstacles can be removed on the path to perfect Web Service execution environment.

Improving error handling in DHArch is a very tempting research issue. Errors can happen and an exception possibly occurs during a message processing. DHArch utilizes a simple approach for error handling; when an error occurs, the message processing is halted and the error is propagated back to the requester. A stateless handler is an exception for this policy. DHArch internal reliable mechanism repeats the execution for the stateless handler. This behavior prevents the execution starting from the beginning. On the other hand, the same policy cannot be applied for the stateful handlers. Handlers must either be atomic or introduce new mechanisms so that the execution does not lead to inconsistency. Check points can be applied not to start the service execution from the beginning without causing any inconsistency.

Another type of future work is to find out the best handler deployment configuration. The distribution of the handlers puts many choices in front of us. Because of the parallelism, the handler orchestration can be achieved in many ways. However, the throughput cannot be increased by a randomly selected handler sequence. Having an agent that intelligently looks for a better handler orchestration sequence is a very promising research area. This agent automates the handler orchestration and adjusts the handler sequence for the best throughput.

Load balancing is another interesting research area. Handlers are being able to be replicated in DHArch. However, one instance can work at a moment for a message. The running replica is selected according to its priority. However, the replicated handlers running simultaneously would perform better.

Finally, DHArch separates the description from execution for the handler orchestration. Having this mechanism opens a door for a promising area. The description level provides an environment to utilize Workflow and Orchestration tools. They can be leveraged to simulate and/or manage the handler executions.

Appendix A

Handler Orchestration Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
        <!--Element Definitions-->
        <xs:element name="name" type="xs:string"/>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="oneway" type="xs:boolean"/>
        <xs:element name="mustPerform" type="xs:boolean"/>
        <xs:element name="condition" type="xs:anyType"/>
        <xs:element name="numberOfHandler" type="xs:short"/>
        <xs:element name="numberOfLooping" type="xs:short"/>
        <xs:complexType name="timeType">
                <xs:sequence>
                        <xs:element name="definition" type="xs:string"/>
                        <xs:element name="timeElement" type="xs:long"/>
                </xs:sequence>
        </xs:complexType>
        <!--Defines Handler-->
        <xs:complexType name="handlerType">
                <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="address"/>
                        <xs:element ref="mustPerform"/>
                        <xs:element ref="oneway"/>
                        <xs:element name="time" type="timeType" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="handler" type="handlerType"/>
        <!--Defines the types of parallel execution-->
        <xs:element name="typeOfParallelExecution">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:enumeration value="synch"/>
                                <xs:enumeration value="asynch"/>
                        </xs:restriction>
```

```
            </xs:simpleType>
        </xs:element>
        <!--Defines the four execution constructs-->
        <xs:element name="sequential">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="handler" maxOccurs="unbounded"/>
                    <xs:element ref="numberOfHandler"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="parallel">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="handler" maxOccurs="unbounded"/>
                    <xs:element ref="numberOfHandler"/>
                    <xs:element ref="typeOfParallelExecution"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="conditional">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="handler" maxOccurs="unbounded"/>
                    <xs:element ref="condition"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="looping">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="handler"/>
                    <xs:element ref="numberOfLooping"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <!--Defines the execution consturct itself-->
        <xs:element name="executionConstruct">
            <xs:complexType>
                <xs:choice>
                    <xs:element ref="sequential"/>
                    <xs:element ref="parallel"/>
                    <xs:element ref="looping"/>
                    <xs:element ref="conditional"/>
                </xs:choice>
                <xs:attribute name="position" type="xs:short" use="required"/>
```

```
                </xs:complexType>
        </xs:element>
        <xs:element name="flowSequence">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="executionConstruct"
maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="numberOfCunstruct" type="xs:short"
use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="handlerOrchestration">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="flowSequence"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Appendix B

Policy Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:element name="name" type="xs:string"/>
	<xs:element name="definition" type="xs:anyType"/>
	<xs:element name="address" type="xs:string"/>
	<xs:element name="oneway" type="xs:boolean"/>
	<xs:element name="numberOfHandler" type="xs:short"/>
	<xs:element name="mustPerform" type="xs:boolean"/>
	<xs:complexType name="timeType">
		<xs:sequence>
			<xs:element name="definition" type="xs:string"/>
			<xs:element name="timeElement" type="xs:long"/>
		</xs:sequence>
	</xs:complexType>
	<!--Defines Handler-->
	<xs:complexType name="handlerType">
		<xs:sequence>
			<xs:element ref="name"/>
			<xs:element ref="address"/>
			<xs:element ref="mustPerform"/>
			<xs:element ref="oneway"/>
			<xs:element name="time" type="timeType" minOccurs="0"
maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
	<xs:element name="handler" type="handlerType"/>
	<xs:element name="type">
		<xs:simpleType>
			<xs:restriction base="xs:string">
				<xs:enumeration value="mustApplied"/>
				<xs:enumeration value="optional"/>
			</xs:restriction>
		</xs:simpleType>
	</xs:element>
```

```
        <xs:element name="orchestrationSchema">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="fileName"/>
                                <xs:element name="Version"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="orderRestriction">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="type"/>
                                <xs:element ref="handler" maxOccurs="unbounded"/>
                                <xs:element ref="numberOfHandler"/>
                        </xs:sequence>
                        <xs:attribute name="restrictionNumber" type="xs:short"
use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="description">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="type"/>
                                <xs:element ref="definition"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="policy">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="orchestrationSchema"/>
                                <xs:element ref="description" minOccurs="0"
maxOccurs="unbounded"/>
                                <xs:element ref="orderRestriction" minOccurs="0"
maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Appendix C

An instance of the Handler Orchestration Document

```
<?xml version="1.0" encoding="UTF-8"?>
<handlerOrchestration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\research_doc\thesis\chapters\architecture\workf
low\final_flow_schema.xsd">
        <flowSequence numberOfCunstruct="4">
                <executionConstruct position="1">
                        <sequential>
                                <handler>
                                        <name>handler 1</name>
                                        <address>/dharch/handler1</address>
                                        <mustPerform>true</ mustPerform >
                                        <oneway>false</oneway>

                                </handler>
                                <handler>
                                        <name>handler 2</name>
                                        <address>/dharch/handler2</address>
                                        <mustPerform>true</ mustPerform >
                                        <oneway>false</oneway>
                                </handler>
                                <handler>
                                        <name>handler 3</name>
                                        <address>/dharch/handler3</address>
                                        <mustPerform>true</ mustPerform >
                                        <oneway>true</oneway>
                                </handler>
                                <numberOfHandler>3</numberOfHandler>
                        </sequential>
                </executionConstruct>
                <executionConstruct position="2">
                        <parallel>
                                <handler>
                                        <name>handler 4</name>
                                        <address>/dharch/handler4</address>
                                        <mustPerform>true</ mustPerform >
                                        <oneway>false</oneway>
```

```
                    </handler>
                    <handler>
                        <name>handler 5</name>
                        <address>/dharch/handler5</address>
                        <mustPerform>true</ mustPerform >
                        <oneway>false</oneway>
                    </handler>
                    <handler>
                        <name>handler 6</name>
                        <address>/dharch/handler6</address>
                        <mustPerform>true</ mustPerform >
                        <oneway>false</oneway>

                    </handler>
                    <handler>
                        <name>handler 7</name>
                        <address>/dharch/handler7</address>
                        <mustPerform>true</ mustPerform >
                        <oneway>false</oneway>
                    </handler>
                    <numberOfHandler>4</numberOfHandler>

<typeOfParallelExecution>synch</typeOfParallelExecution>
                </parallel>
            </executionConstruct>
            <executionConstruct position="3">
                <looping>
                    <handler>
                        <name>handler 8</name>
                        <address>/dharch/handler8</address>
                        <mustPerform>true</ mustPerform >
                        <oneway>false</oneway>

                    </handler>
                    <numberOfLooping>2</numberOfLooping>
                </looping>
            </executionConstruct>
            <executionConstruct position="4">
                <conditional>
                    <handler>
                        <name>handler 9</name>
                        <address>/dharch/handler10</address>
                        <mustPerform>true</ mustPerform >
                        <oneway>true</oneway>
                    </handler>
                    <handler>
```

```
                                        <name>handler 10</name>
                                        <address>/dharch/handler10</address>
                                        <mustPerform>true</ mustPerform >
                                        <oneway>false</oneway>
                                </handler>
                                <condition>
                                        <isElementExist elementName="wsLog">handler
9</isElementExist>
                                </condition>
                        </conditional>
                </executionConstruct>
        </flowSequence>
</handlerOrchestration>
```

Appendix D

Web Service specifications and the SOAP parts being interested

Specification Name	SOAP Part header or body
WS-ReliableMessaging	Both
WS-Reliability	Both
WS-Addressing	Both[3]
WS-Security	Both
WSS:SOAP Message Security	Both
WSS:UsernameTaken Profile	Header
WSS:X.509 Certificate Token Profile	Both[4]
WSS:Kerberos Binding	Both
WS-Security Addendum	Both
WS-Trust	Body
WS-SecureConversation	Both
WS-Notification	Body [5]
WS-BaseNotification	Body
WS-Topic	Body[6]
WS-BrokeredNotification	Body
WS-Policy	Both [7]
WS-SecurityPolicy	header[8]
WS-PolicyAssertions	Both
WS-PolicyAttachment	Both
WS-MetadataExchange	Body
WS-ResourceFramework	Body[9]
WS-ResourceProperties	Body
WS-ResourceLifetime	Body

[3] Although the namespace appears mostly header, it may appear in the body too.
[4] Header consists of the related information with X.509. Modification of the body is happened because of encryption.
[5] WS-Notification is used to refer family of specifications. This family consists WS-BaseNotification, WS-Topic, WS-BrokeredNotification. WS-Resource Framework family, WS-Addressing, WS-Security, WS-SecureConversation, WS-Trust may also contribute.
[6]. In this specification, it is not mentioned whether WS-Topic is used in body or header. Since it is used by WS-Notification, it must be in body.
[7] It can be seen in the body with WS-MetadataExchange.
[8] Since it uses WS-Policy and WS-Security, WS-SecurityPolicy may modify both header and body.
[9] Includes other specifications,WS-ResourceProperties,WS-ResourceLifetime,WS-BaseFault,WS-ServiceGroup

WS-BaseFaults	[10]
WS-ServiceGroup	Body
WS-Routing	header
WS-Referral	header
WS-Federation	Both[11]
WS-Active-Profile	Both[12]
WS-Passive-Profile	Both[13]
WS-Discovery	Body
WS-Provisioning	Body
WS-Enumeration	Body
WS-Eventing	Both[14]
WS-Transfer	Both[15]
WS-Inspection	[16]
WS-Management	Both
WS-Coordination	Header
WS-Transaction	Header[17]
WS-AtomicTransection	Header[18]
WS-BusinessActivity	Header[19]
WS-Attachment	[20]
BPELWS	[21]
WS-I	Both[22]

[10] WS-BaseFaults defines an XML Schema type for a base fault, along with rules for how this fault type is used by Web services.
[11] WS- Federation describes the overall model for authentication which builds on the foundations specified in WS-Security, WS-Policy, and WS-Trust.
[12] The federation model described in WS-Federation builds on the foundation established by WS-Security and WS-Trust. Consequently, this profile defines mechanisms for requesting, exchanging, and issuing security tokens within the context of active requestor.
[13] The federation model described in WS-Federation builds on the foundation established by WS-Security and WS-Trust. Consequently, this specification profiles the mechanisms for requesting, exchanging, and issuing security tokens within the context of a passive requestor.
[14] The modification in header is done by WS-Addressing
[15] Although WS-Transfer tag appears neither header nor body, some elements are added to both header and body.
[16] WS-Inspection document is nothing more than an aggregation of pointers to service description documents. It is not related with neither SOAP header nor SOAP body
[17] By using the SOAP and WSDL extensibility model, SOAP-based and WSDL-based specifications are designed to work together to define a rich web services environment. As such, WS-Transaction by itself does not define all features required for a complete solution. WS-Transaction is a building block used with other specifications of web services (e.g., WS-Coordination, WS-Security) and application-specific protocols that are able to accommodate a wide variety of coordination protocols related to the coordination actions of distributed applications. There are two coordination types; Atomic Transaction and Business Activity
[18] This specification provides the definition of the atomic transaction coordination type that is to be used with the extensible coordination framework described in the WS-Coordination specification
[19] This specification provides the definition of the business activity coordination type that is to be used with the extensible coordination framework described in the WS-Coordination specification
[20] There may be URI reference, which is added to body or header, to the attachment.
[21] Extends WSDL

WS-CAF	Both[23]
WS-Context	Header[24]
Web Service Coordination Framework WS-CF	Header[25]
Web Service Transaction Management WS-TXM	Header[26]
UDDI	Body

[22] It uses other specifications. They can be divided into two part; Basic Profiles and Additional Profiles. Basic profiles include XML Schema, SOAP, WSDL and UDDI. Since more specifications are needed to make web services interoperable, other specifications are used in WS-I such as security inspection and discovery.

[23] The WS-CAF is divided into three parts; WS-TXM, WS-CTX and WS-CF. In this specification, Web services can also choose to join a composite application upon receipt of a SOAP message containing the context URI in the header, or, optionally, containing the context itself within the body of the SOAP message.

[24] "Context is always propagated in addition to application payload, where context information travels within the SOAP header blocks while application payload in the body"

[25] "All operations on the coordinator service are implicitly associated with the current context". To do so, it uses extended context mechanism. It also adds several portTypes in order to manage coordination.

[26] WS-TXM builds on the Web Services Coordination Framework (WS-CF) and Web Service CTX Service (WS-CTX) specifications. It does this by defining specific coordinator and participant services (portTypes) and augmenting the distribution context.

Appendix E

SOAP messages for WS-Resource Framework

WSDL document of Web Service Resource Framework for Sensor

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://ws.apache.org/resource/example/sensor"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-
1.2-draft-01.wsdl" xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.wsdl" name="SensorResourceDefinition"
targetNamespace="http://ws.apache.org/resource/example/sensor">
        <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl" location="../spec/wsrf/WS-ResourceProperties-
1_2-Draft_01.wsdl"/>
        <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.wsdl" location="../spec/wsrf/WS-ResourceLifetime-
1_2-Draft_01.wsdl"/>
        <types>
                <schema elementFormDefault="qualified"
targetNamespace="http://ws.apache.org/resource/example/sensor"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsrl="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd"
xmlns:wsbf="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-
01.xsd">
                        <xsd:import namespace="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-01.xsd"
schemaLocation="../spec/wsrf/WS-BaseFaults-1_2-Draft_01.xsd"/>
                        <xsd:import namespace="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd"
schemaLocation="../spec/wsrf/WS-ResourceLifetime-1_2-Draft_01.xsd"/>
                        <element name="Type" type="xsd:string"/>
                        <element name="Location" type="xsd:string"/>
                        <element name="LastTimeOfSignal" type="xsd:string"/>
                        <element name="Options">
```

```
                        <complexType>
                            <sequence>
                                <element name="Option"
type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
                            </sequence>
                        </complexType>
                    </element>
                    <element name="SignalFrequency" type="xsd:int"/>
                    <element name="StartedTime" type="xsd:string"/>
                    <element name="Comment" type="xsd:string"/>
                    <!-- Resource Properties Document Schema -->
                    <element name="SensorProperties">
                        <complexType>
                            <sequence>
                                <!-- props for
wsrl:ScheduledResourceTermination portType -->
                                <element ref="wsrl:CurrentTime"/>
                                <element ref="wsrl:TerminationTime"/>
                                <!-- props for tns:SensorPortType
portType -->
                                <element ref="tns:Type"/>
                                <element ref="tns:Location"/>
                                <element ref="tns:LastTimeOfSignal"/>
                                <element ref="tns:Options"/>
                                <element ref="tns:SignalFrequency"/>
                                <element ref="tns:StartedTime"/>
                                <element ref="tns:Comment"
minOccurs="0"/>
                            </sequence>
                        </complexType>
                    </element>
                    <!-- ====== Message Types for Custom Operations  ======= -
->
                    <element name="Start">
                        <complexType/>
                    </element>
                    <element name="StartResponse">
                        <complexType/>
                    </element>
                    <element name="Stop">
                        <complexType/>
                    </element>
                    <element name="StopResponse">
                        <complexType/>
                    </element>
                    <element name="DeviceBusyFault">
```

```
                        <complexType>
                            <complexContent>
                                <extension base="wsbf:BaseFaultType"/>
                            </complexContent>
                        </complexType>
                    </element>
                </schema>
            </types>
            <message name="StartRequest">
                <part name="StartRequest" element="tns:Start"/>
            </message>
            <message name="StartResponse">
                <part name="StartResponse" element="tns:StartResponse"/>
            </message>
            <message name="StopRequest">
                <part name="StopRequest" element="tns:Stop"/>
            </message>
            <message name="StopResponse">
                <part name="StopResponse" element="tns:StopResponse"/>
            </message>
            <message name="DeviceBusyFault">
                <part name="DeviceBusyFault" element="tns:DeviceBusyFault"/>
            </message>
            <portType name="SensorPortType"
wsrp:ResourceProperties="tns:SensorProperties">
                <!-- wsrp:* operations -->
                <operation name="GetResourceProperty">
                    <input name="GetResourcePropertyRequest"
message="wsrpw:GetResourcePropertyRequest"/>
                    <output name="GetResourcePropertyResponse"
message="wsrpw:GetResourcePropertyResponse"/>
                    <fault name="ResourceUnknownFault"
message="wsrpw:ResourceUnknownFault"/>
                    <fault name="InvalidResourcePropertyQNameFault"
message="wsrpw:InvalidResourcePropertyQNameFault"/>
                </operation>
                <operation name="GetMultipleResourceProperties">
                    <input name="GetMultipleResourcePropertiesRequest"
message="wsrpw:GetMultipleResourcePropertiesRequest"/>
                    <output name="GetMultipleResourcePropertiesResponse"
message="wsrpw:GetMultipleResourcePropertiesResponse"/>
                    <fault name="ResourceUnknownFault"
message="wsrpw:ResourceUnknownFault"/>
                    <fault name="InvalidResourcePropertyQNameFault"
message="wsrpw:InvalidResourcePropertyQNameFault"/>
                </operation>
```

```
                <operation name="SetResourceProperties">
                        <input name="SetResourcePropertiesRequest"
message="wsrpw:SetResourcePropertiesRequest"/>
                        <output name="SetResourcePropertiesResponse"
message="wsrpw:SetResourcePropertiesResponse"/>
                        <fault name="ResourceUnknownFault"
message="wsrpw:ResourceUnknownFault"/>
                        <fault name="InvalidResourcePropertyQNameFault"
message="wsrpw:InvalidResourcePropertyQNameFault"/>
                        <fault
name="InvalidSetResourcePropertiesRequestContentFault"
message="wsrpw:InvalidSetResourcePropertiesRequestContentFault"/>
                        <fault name="UnableToModifyResourcePropertyFault"
message="wsrpw:UnableToModifyResourcePropertyFault"/>
                        <fault name="SetResourcePropertyRequestFailedFault"
message="wsrpw:SetResourcePropertyRequestFailedFault"/>
                </operation>
                <operation name="QueryResourceProperties">
                        <input name="QueryResourcePropertiesRequest"
message="wsrpw:QueryResourcePropertiesRequest"/>
                        <output name="QueryResourcePropertiesResponse"
message="wsrpw:QueryResourcePropertiesResponse"/>
                        <fault name="ResourceUnknownFault"
message="wsrpw:ResourceUnknownFault"/>
                        <fault name="InvalidResourcePropertyQNameFault"
message="wsrpw:InvalidResourcePropertyQNameFault"/>
                        <fault name="UnknownQueryExpressionDialectFault"
message="wsrpw:UnknownQueryExpressionDialectFault"/>
                        <fault name="InvalidQueryExpressionFault"
message="wsrpw:InvalidQueryExpressionFault"/>
                        <fault name="QueryEvaluationErrorFault"
message="wsrpw:QueryEvaluationErrorFault"/>
                </operation>
                <!-- wsrl:ImmediateResourceTermination operation -->
                <operation name="Destroy">
                        <input message="wsrlw:DestroyRequest"/>
                        <output message="wsrlw:DestroyResponse"/>
                        <fault name="ResourceUnknownFault"
message="wsrlw:ResourceUnknownFault"/>
                        <fault name="ResourceNotDestroyedFault"
message="wsrlw:ResourceNotDestroyedFault"/>
                </operation>
                <!-- wsrl:ScheduledResourceTermination operation -->
                <operation name="SetTerminationTime">
                        <input message="wsrlw:SetTerminationTimeRequest"/>
                        <output message="wsrlw:SetTerminationTimeResponse"/>
```

```
                        <fault name="ResourceUnknownFault"
message="wsrlw:ResourceUnknownFault"/>
                        <fault name="UnableToSetTerminationTimeFault"
message="wsrlw:UnableToSetTerminationTimeFault"/>
                        <fault name="TerminationTimeChangeRejectedFault"
message="wsrlw:TerminationTimeChangeRejectedFault"/>
                </operation>
                <!-- custom operations -->
                <operation name="Start">
                        <input name="StartRequest" message="tns:StartRequest"/>
                        <output name="StartResponse" message="tns:StartResponse"/>
                        <fault name="DeviceBusyFault"
message="tns:DeviceBusyFault"/>
                </operation>
                <operation name="Stop">
                        <input name="StopRequest" message="tns:StopRequest"/>
                        <output name="StopResponse" message="tns:StopResponse"/>
                        <fault name="DeviceBusyFault"
message="tns:DeviceBusyFault"/>
                </operation>
        </portType>
        <binding name="SensorSoapHttpBinding" type="tns:SensorPortType">
                <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
                <!-- wsrp:* operations -->
                <operation name="GetResourceProperty">
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="ResourceUnknownFault">
                                <soap:fault name="ResourceUnknownFault"
use="literal"/>
                        </fault>
                        <fault name="InvalidResourcePropertyQNameFault">
                                <soap:fault
name="InvalidResourcePropertyQNameFault" use="literal"/>
                        </fault>
                </operation>
                <operation name="GetMultipleResourceProperties">
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
```

```
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="ResourceUnknownFault">
                                <soap:fault name="ResourceUnknownFault"
use="literal"/>
                        </fault>
                        <fault name="InvalidResourcePropertyQNameFault">
                                <soap:fault
name="InvalidResourcePropertyQNameFault" use="literal"/>
                        </fault>
                </operation>
                <operation name="SetResourceProperties">
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="ResourceUnknownFault">
                                <soap:fault name="ResourceUnknownFault"
use="literal"/>
                        </fault>
                        <fault name="InvalidResourcePropertyQNameFault">
                                <soap:fault
name="InvalidResourcePropertyQNameFault" use="literal"/>
                        </fault>
                        <fault name="UnableToModifyResourcePropertyFault">
                                <soap:fault
name="UnableToModifyResourcePropertyFault" use="literal"/>
                        </fault>
                        <fault
name="InvalidSetResourcePropertiesRequestContentFault">
                                <soap:fault
name="InvalidSetResourcePropertiesRequestContentFault" use="literal"/>
                        </fault>
                        <fault name="SetResourcePropertyRequestFailedFault">
                                <soap:fault
name="SetResourcePropertyRequestFailedFault" use="literal"/>
                        </fault>
                </operation>
                <operation name="QueryResourceProperties">
                        <soap:operation style="document"/>
                        <input>
```

```
                        <soap:body use="literal"/>
                    </input>
                    <output>
                        <soap:body use="literal"/>
                    </output>
                    <fault name="ResourceUnknownFault">
                        <soap:fault name="ResourceUnknownFault"
use="literal"/>
                    </fault>
                    <fault name="InvalidResourcePropertyQNameFault">
                        <soap:fault
name="InvalidResourcePropertyQNameFault" use="literal"/>
                    </fault>
                    <fault name="UnknownQueryExpressionDialectFault">
                        <soap:fault
name="UnknownQueryExpressionDialectFault" use="literal"/>
                    </fault>
                    <fault name="InvalidQueryExpressionFault">
                        <soap:fault name="InvalidQueryExpressionFault"
use="literal"/>
                    </fault>
                    <fault name="QueryEvaluationErrorFault">
                        <soap:fault name="QueryEvaluationErrorFault"
use="literal"/>
                    </fault>
                </operation>
                <!-- wsrl:ImmediateResourceTermination operation -->
                <operation name="Destroy">
                    <soap:operation style="document"/>
                    <input>
                        <soap:body use="literal"/>
                    </input>
                    <output>
                        <soap:body use="literal"/>
                    </output>
                    <fault name="ResourceUnknownFault">
                        <soap:fault name="ResourceUnknownFault"
use="literal"/>
                    </fault>
                    <fault name="ResourceNotDestroyedFault">
                        <soap:fault name="ResourceNotDestroyedFault"
use="literal"/>
                    </fault>
                </operation>
                <!-- wsrl:ScheduledResourceTermination operation -->
                <operation name="SetTerminationTime">
```

```
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="ResourceUnknownFault">
                                <soap:fault name="ResourceUnknownFault"
use="literal"/>
                        </fault>
                        <fault name="UnableToSetTerminationTimeFault">
                                <soap:fault name="UnableToSetTerminationTimeFault"
use="literal"/>
                        </fault>
                        <fault name="TerminationTimeChangeRejectedFault">
                                <soap:fault
name="TerminationTimeChangeRejectedFault" use="literal"/>
                        </fault>
                </operation>
                <!-- custom operations -->
                <operation name="Start">
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="DeviceBusyFault">
                                <soap:fault name="DeviceBusyFault" use="literal"/>
                        </fault>
                </operation>
                <operation name="Stop">
                        <soap:operation style="document"/>
                        <input>
                                <soap:body use="literal"/>
                        </input>
                        <output>
                                <soap:body use="literal"/>
                        </output>
                        <fault name="DeviceBusyFault">
                                <soap:fault name="DeviceBusyFault" use="literal"/>
                        </fault>
                </operation>
        </binding>
```

```
        <service name="SensorService">
                <port name="sensor" binding="tns:SensorSoapHttpBinding">
                        <soap:address
location="http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor"/>
                </port>
        </service>
</definitions>
```

Inquiry for sensor data:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:sn="http://ws.apache.org/resource/example/sensor">
  <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
       <wsa:To mustUnderstand="1">http://
gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
       <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/y
ourWsdlRequestName</wsa:Action>
       <fs:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</fs:ResourceIdentifier>
  </Header>
  <Body>
    <wsrp:QueryResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd">
       <wsrp:QueryExpression Dialect="http://www.w3.org/TR/1999/REC-xpath-
19991116">*</wsrp:QueryExpression>
    </wsrp:QueryResourceProperties>
  </Body>
</Envelope>
```

The result of inquiry without any update

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
  <wsa:Action soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmls
oap.org/ws/2004/03/addressing/anonymous</wsa:Action>
  <wsa:To soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmls
oap.org/ws/2004/03/addressing/anonymous</wsa:To>
```

```
 </soapenv:Header>
 <soapenv:Body>
  <wsrf:QueryResourcePropertiesResponse xmlns:wsrf="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd">
   <wsrf:CurrentTime xmlns:wsrf="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.xsd">2007-03-01T00:37:15.117-
05:00</wsrf:CurrentTime>
   <wsrf:TerminationTime xsi:nil="true" xmlns:wsrf="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd"/>
   <sen:Type
xmlns:sen="http://ws.apache.org/resource/example/sensor">/sensor/cal/1</sen:Type>
   <sen:Location
xmlns:sen="http://ws.apache.org/resource/example/sensor">california</sen:Location>
   <sen:Comment xmlns:sen="http://ws.apache.org/resource/example/sensor">very
important</sen:Comment>
   <sen:StartedTime
xmlns:sen="http://ws.apache.org/resource/example/sensor">0</sen:StartedTime>
   <sen:LastTimeOfSignal
xmlns:sen="http://ws.apache.org/resource/example/sensor">Monday February
26</sen:LastTimeOfSignal>
   <sen:SignalFrequency
xmlns:sen="http://ws.apache.org/resource/example/sensor">5</sen:SignalFrequency>
   <sen:Options xmlns:sen="http://ws.apache.org/resource/example/sensor">
    <sen:Option>name</sen:Option>
    <sen:Option>number</sen:Option>
   </sen:Options>
  </wsrf:QueryResourcePropertiesResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Messages to update the sensor data

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:sn="http://ws.apache.org/resource/example/sensor">
   <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
       <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
       <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/y
ourWsdlRequestName</wsa:Action>
       <sn:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
   </Header>
   <Body>
     <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
```

```
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
      <wsrp:Update>
        <sn:Comment>this sensor is very important to analyze .... </sn:Comment>
      </wsrp:Update>
    </wsrp:SetResourceProperties>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:sn="http://ws.apache.org/resource/example/sensor">
   <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
      <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
      <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/y
ourWsdlRequestName</wsa:Action>
      <sn:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
  </Header>
  <Body>
    <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
      <wsrp:Update>
        <sn:LastTimeOfSignal>10:20:32 AM February
23,2007</sn:LastTimeOfSignal>
      </wsrp:Update>
    </wsrp:SetResourceProperties>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:sn="http://ws.apache.org/resource/example/sensor">

   <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
      <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
      <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/y
ourWsdlRequestName</wsa:Action>
      <sn:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
  </Header>
```

```
  <Body>
    <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
      <wsrp:Update>
        <sn:Options>
        <sn:Option>Do we need to restart this?</sn:Option>
        <sn:Option>yes</sn:Option>
        <sn:Option>Do we need to keep previous month data?</sn:Option>
        <sn:Option>no</sn:Option>
        <sn:Option>Is it necessary to inform the people if abnormal activity is
observed?</sn:Option>
        <sn:Option>yes</sn:Option>
        </sn:Options>
      </wsrp:Update>
    </wsrp:SetResourceProperties>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:sn="http://ws.apache.org/resource/example/sensor">

   <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
      <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
      <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/y
ourWsdlRequestName</wsa:Action>
      <sn:ResourceIdentifier
mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
  </Header>

  <Body>
    <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
      <wsrp:Update>
        <sn:SignalFrequency>10</sn:SignalFrequency>
      </wsrp:Update>
    </wsrp:SetResourceProperties>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
```

```
        xmlns:sn="http://ws.apache.org/resource/example/sensor">

   <Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
       <wsa:To
mustUnderstand="1">http://gf4.ucs.indiana.edu:8080/wsrf/services/sensor</wsa:To>
       <wsa:Action
mustUnderstand="1">http://ws.apache.org/resource/example/sensor/SensorPortType/you
rWsdlRequestName</wsa:Action>
       <sn:ResourceIdentifier mustUnderstand="1">/sensor/cal/1</sn:ResourceIdentifier>
   </Header>

   <Body>
     <wsrp:SetResourceProperties xmlns:wsrp="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
                        xmlns:sn="http://ws.apache.org/resource/example/sensor">
       <wsrp:Update>
         <sn:StartedTime>Wednesday, February 7,2007</sn:StartedTime>
       </wsrp:Update>
     </wsrp:SetResourceProperties>
   </Body>
</Envelope>
```

The response for an update request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
  <wsa:Action soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoa
p.org/ws/2004/03/addressing/anonymous</wsa:Action>
  <wsa:To soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoa
p.org/ws/2004/03/addressing/anonymous</wsa:To>
 </soapenv:Header>
 <soapenv:Body>
  <wsrf:SetResourcePropertiesResponse xmlns:wsrf="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"/>
 </soapenv:Body>
</soapenv:Envelope>
```

Inquiry after updates

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
  <wsa:Action soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoa
p.org/ws/2004/03/addressing/anonymous</wsa:Action>
  <wsa:To soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoa
p.org/ws/2004/03/addressing/anonymous</wsa:To>
 </soapenv:Header>
 <soapenv:Body>
  <wsrf:QueryResourcePropertiesResponse xmlns:wsrf="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd">
   <wsrf:CurrentTime xmlns:wsrf="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.xsd">2007-03-01T00:41:14.856-
05:00</wsrf:CurrentTime>
   <wsrf:TerminationTime xsi:nil="true" xmlns:wsrf="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd"/>
   <sen:Type
xmlns:sen="http://ws.apache.org/resource/example/sensor">/sensor/cal/1</sen:Type>
   <sen:Location
xmlns:sen="http://ws.apache.org/resource/example/sensor">california</sen:Location>
   <sn:SignalFrequency xmlns:sn="http://ws.apache.org/resource/example/sensor"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-
draft-01.xsd">10</sn:SignalFrequency>
   <sn:StartedTime xmlns:sn="http://ws.apache.org/resource/example/sensor"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-
draft-01.xsd">Wednesday, February 7,2007</sn:StartedTime>
   <sn:LastTimeOfSignal xmlns:sn="http://ws.apache.org/resource/example/sensor"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-
draft-01.xsd">10:20:32 AM February 23,2007</sn:LastTimeOfSignal>
   <sn:Comment xmlns:sn="http://ws.apache.org/resource/example/sensor"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-
draft-01.xsd">this sensor is very important to analyze .... </sn:Comment>
   <sn:Options xmlns:sn="http://ws.apache.org/resource/example/sensor"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-
draft-01.xsd">
          <sn:Option>Do we need to restart this?</sn:Option>
           <sn:Option>yes</sn:Option>
           <sn:Option>Do we need to keep previous month data?</sn:Option>
           <sn:Option>no</sn:Option>
           <sn:Option>Is it necessary to inform the people if abnormal activity is
observed?</sn:Option>
```

```
            <sn:Option>yes</sn:Option>
            <sn:Option>name</sn:Option>
           <sn:Option>number</sn:Option>
    </sn:Options>
 </wsrf:QueryResourcePropertiesResponse>
</soapenv:Body></soapenv:Envelope>
```

Appendix F

SOAP messages for WS-Eventing

Sink subscription request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe</add:Action>
    <add:MessageID>82678a00-5da4-4648-8758-fa02b259d48e</add:MessageID>
    <add:From>
      <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
    </add:From>
    <add:To>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:To>
  </Header>
  <Body>
    <even:Subscribe>
      <even:EndTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
      </even:EndTo>
      <even:Delivery
Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
      <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
      <even:Filter
Dialect="http://www.naradabrokering.org/TopicMatching">/sensor/cal</even:Filter>
      <even:NotifyTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
      </even:NotifyTo>
    </even:Subscribe>
  </Body>
</Envelope>
```

Created SOAP message in Subscription Manager for the request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/SubscribeResponse</add:
Action>
   <add:MessageID>caf16ae1-e4eb-40b6-bbf7-862c47438919</add:MessageID>
   <add:From>

<add:Address>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:Address>
   </add:From>
   <add:To>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:To>
 </Header>
 <Body>
   <even:Subscribe xmlns="http://schemas.xmlsoap.org/soap/envelope/">
     <even:EndTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
     </even:EndTo>
     <even:Delivery
Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
     <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
     <even:Filter Dialect="http://www.naradabrokering.org/TopicMatching">/sensor/cal
</even:Filter>
     <even:NotifyTo>
<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
     </even:NotifyTo>
   </even:Subscribe>
   <even:SubscribeResponse>
     <even:SubscriptionManager>
<add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
       <add:ReferenceParameters>
         <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>
       </add:ReferenceParameters>
     </even:SubscriptionManager>
     <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
   </even:SubscribeResponse>
```

```
  </Body>
</Envelope>
```

The response received by Sink from Subscription Manager

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/SubscribeResponse</add:
Action>
    <add:RelatesTo>82678a00-5da4-4648-8758-fa02b259d48e</add:RelatesTo>
    <add:MessageID>d649ec2c-508c-4918-a174-1069aa870277</add:MessageID>
    <add:From>

<add:Address>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:Address>
    </add:From>
    <add:To>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:To>
  </Header>
  <Body>
    <even:SubscribeResponse>
      <even:SubscriptionManager>

<add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
        <add:ReferenceParameters>
          <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>
        </add:ReferenceParameters>
      </even:SubscriptionManager>
      <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
    </even:SubscribeResponse>
  </Body>
</Envelope>
```

Source agreement for the subscription

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>
```

```
<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/SubscribeResponse</add:
Action>
   <add:MessageID>caf16ae1-e4eb-40b6-bbf7-862c47438919</add:MessageID>
   <add:From>

<add:Address>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:Address>
   </add:From>
   <add:To>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:To>
  </Header>
  <Body>
   <even:Subscribe xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <even:EndTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
     </even:EndTo>
     <even:Delivery
Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
     <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
     <even:Filter
Dialect="http://www.naradabrokering.org/TopicMatching">/Literature/Shakespere</even
:Filter>
     <even:NotifyTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
     </even:NotifyTo>
   </even:Subscribe>
   <even:SubscribeResponse>
     <even:SubscriptionManager>

<add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
       <add:ReferenceParameters>
         <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>
       </add:ReferenceParameters>
     </even:SubscriptionManager>
     <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
   </even:SubscribeResponse>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe</add:Action>
    <add:MessageID>82678a00-5da4-4648-8758-fa02b259d48e</add:MessageID>
    <add:From>
      <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
    </add:From>
    <add:To>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:To>
  </Header>
  <Body>
    <even:Subscribe>
      <even:EndTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
      </even:EndTo>
      <even:Delivery
Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
      <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
      <even:Filter Dialect="http://www.naradabrokering.org/TopicMatching">/sensor/cal
</even:Filter>
      <even:NotifyTo>

<add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
      </even:NotifyTo>
    </even:Subscribe>
  </Body>
</Envelope>
```

Renewing the lease to increase subscription duration

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>
    <add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Renew</add:Action>
    <add:MessageID>41c86f95-ea4b-43b1-83a8-c44b1cc76e76</add:MessageID>
    <add:From>
```

```
    <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
   </add:From>
   <add:To>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:To>
   <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>
  </Header>
  <Body>
   <even:Renew>
    <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-05-02T22:17:04.933-
04:00</even:Expires>
   </even:Renew>
  </Body>
</Envelope>
```

Response to Sink for renewal from WseSM

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/RenewResponse</add:Ac
tion>
   <add:RelatesTo>41c86f95-ea4b-43b1-83a8-c44b1cc76e76</add:RelatesTo>
   <add:MessageID>600ac0a3-3d8c-4a7b-8e57-aaff2f887a91</add:MessageID>
   <add:From>
    <add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
   </add:From>
   <add:To>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:To>
  </Header>
  <Body>
   <even:RenewResponse>
    <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-05-02T22:17:04.933-
04:00</even:Expires>
   </even:RenewResponse>
  </Body>
</Envelope>
```

Renewal message to Source from Subscription Manager

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
  <Header>
    <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/RenewResponse</add:Ac
tion>
    <add:MessageID>7c510060-de40-49d8-ae5c-73dd768fa652</add:MessageID>
    <add:From>
      <add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
    </add:From>
    <add:To>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:To>
  </Header>
  <Body>
    <even:RenewResponse>
      <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-05-02T22:17:04.933-
04:00</even:Expires>
    </even:RenewResponse>
  </Body>
</Envelope>
```

Unsubscribe

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Unsubscribe</add:Action
>
    <add:MessageID>c2cc676c-d362-4fa4-a04e-4618175c9445</add:MessageID>
    <add:From>
      <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
    </add:From>
    <add:To>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:To>
    <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9ac</even:Identifier>
  </Header>
  <Body>
    <even:Unsubscribe/>
  </Body>
</Envelope>
```

Getting status

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
```

```
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/GetStatus</add:Action>
    <add:MessageID>85402a7f-99ed-40e7-a3a0-ca9eb0580c58</add:MessageID>
    <add:From>
      <add:Address>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:Address>
    </add:From>
    <add:To>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:To>
    <even:Identifier>e23c08b5-7622-4b4d-98d2-a765fe1c9acb</even:Identifier>
  </Header>
  <Body>
    <even:GetStatus/>
  </Body>
</Envelope>
```

The response for status request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Header>

<add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/GetStatusResponse</add:
Action>
    <add:RelatesTo>85402a7f-99ed-40e7-a3a0-ca9eb0580c58</add:RelatesTo>
    <add:MessageID>61a587d9-c6c1-4c0f-acc7-ed40d8b4bb64</add:MessageID>
    <add:From>
      <add:Address>http://gf5.ucs.indiana.edu:10080/axis/services/WseSM</add:Address>
    </add:From>
    <add:To>http://gf3.ucs.indiana.edu:9080/axis/services/WseSink</add:To>
  </Header>
  <Body>
    <even:GetStatusResponse>
      <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2007-04-02T22:02:19.495-
04:00</even:Expires>
    </even:GetStatusResponse>
  </Body>
</Envelope>
```

The message being sent by Source and received by Sink

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:top="http://www.naradabrokering.org/TopicMatching"
```

```
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:sens="http://www.naradabrokering.org/sensor"
xmlns:nar="http://www.naradabrokering.org">
  <Header>
    <top:Topic>/sensor/cal</top:Topic>
    <add:MessageID>c3e00553-db0c-4ae0-965a-a59183ed3761</add:MessageID>
    <add:From>

<add:Address>http://gf4.ucs.indiana.edu:8080/axis/services/WseSource</add:Address>
    </add:From>
  </Header>
  <Body>
    <sens:sensor>
      <sens:cal>
        <sens:number>1</sens:number>
        <sens:CurrentTime>2007-03-01T00:41:14.856-05:00</sens:CurrentTime>
        <sens:Location>california</sens:Location>
        <nar:Application-Content>Tracker 1 : Important activity happend</nar:Application-
Content>
      </sens:cal>
    </sens:sensor>
  </Body>
</Envelope>
```

Bibliography

1. Web Service Architecture, http://www.w3.org/TR/ws-arch/.
2. Simple Object Access Protocol (SOAP), http://www.w3.org/TR/soap12-part1/.
3. Web Service Description Language (WSDL), http://www.w3.org/TR/wsdl.
4. Universal Description Discovery and Integration (UDDI), http://www.uddi.org/ .
5. Apache Axis, http://ws.apache.org/axis/.
6. Microsoft Web Service Enhancements (WSE), http://www.microsoft.com/downloads/details.aspx?FamilyId=FC5F06C5-821F-41D3-A4FE-6C7B56423841&displaylang=en.
7. IBM WebSphere, http://www-306.ibm.com/software/websphere/.
8. Web Service Specifications, http://www-128.ibm.com/developerworks/webservices/library/ws-spec.html.
9. Hoare, C.A.R., *The emperor's old clothes*. 1981, ACM Press New York, NY, USA. p. 75-83.
10. Lundstrom, M., *APPLIED PHYSICS: Enhanced: Moore's Law Forever?* Science 2003 Vol. 299. no. 5604, pp. 210 - 211.
11. Tran, P., Greenfield, P., and Gorton, I., *Behavior and Performance of Message-Oriented Middleware Systems.* . Proceedings of the 22nd international Conference on Distributed Computing Systems, ICDCSW. 2002.
12. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. , *1999 Hypertext Transfer Protocol -- Http/1.1.* . RFC. RFC Editor.

13. Pallickara, S. and G. Fox, *NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids*. 2003, Springer.

14. Apache Axis2, http://ws.apache.org/axis2.

15. Apache Tomcat, http://tomcat.apache.org/.

16. Apache WSS4J, An Implementation of WS-Security, http://ws.apache.org/wss4j/.

17. Web Service Security (WS-Security) , http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

18. Apache Sandesha, An Implementation of WS-ReliableMessaging http://ws.apache.org/sandesha/.

19. Web Service Reliable Messaging (WS-ReliableMessaging), ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf.

20. Apache WSRF, An implementation of WS-Resource Framework, http://ws.apache.org/wsrf/.

21. Web Service Recource Framework (WS-Recource Framework), http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf.

22. Web Services Reliability (WS-Reliability) http://www.oracle.com/technology/tech/webservices/htdocs/spec/WS-ReliabilityV1.0.pdf.

23. Web Services Notification (WS-Notification) http://www-106.ibm.com/developerworks/library/specification/ws-notification/.

24. Java, A.P.I., *for XML-Based RPC (JAX-RPC)*. 2003.

25. SOAP with Attachments API for Java (SAAJ), http://java.sun.com/webservices/saaj/index.jsp.

26. Birrell, A.D. and B. Nelson, *Implementing Remote Procedure Calls*, ACM Trans. Comput. Syst. 2, 1 (Feb. 1984), 39-59.

27. Yildiz, B., S. Pallickara, and G. Fox, *Experiences in Deploying Services within Apache Axis Container*, Proceedings of IEEE International Conference on Internet and Web Applications and Services ICIW'06 February 23-25, 2006 Guadeloupe, French Caribbean.

28. Perera, S., C.Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, G. Daniels, *Axis2, Middleware for Next Generation Web Services.* . in IEEE International Conference on Web Services (ICWS'06). 2006.

29. Fry, C., *JSR 173: Streaming API for XML*. 2004.

30. AXIOM Tutorial, Object Module, http://ws.apache.org/axis2/1_0/OMTutorial.html.

31. Web Service Addressing(WS-Addressing), http://www.w3.org/Submission/ws-addressing/.

32. Shrideep Pallickara, et al., *On the Costs for Reliable Messaging in Web/Grid Service Environments.* Proceedings of the 2005 IEEE International Conference on e-Science & Grid Computing. Melbourne, Australia.pp 344-351.

33. Shirasuna, S., et al., *Performance comparison of security mechanisms for grid services*. p. 360-364.

34. Slominski, A., et al., *Asynchronous Peer-to-Peer Web Services and Firewalls*, In 7th International Workshop on Java for Parallel and Distributed Programming (IPDPS 2005), April 2005.

35. Fang, L., A. Slominski, and D. Gannon, *Web Services Security and Load Balancing in Grid Environment*.

36. Ping Guo, et al., *Parsing XML Efficiently.* Oracle, September/October 2003, http://www.oracle.com/technology/oramag/oracle/03-sep/o53devxml.html.

37. Sosnoski, D., *XML and Java technologies: Document models, Part 1: Performance.* IBM, September 2001, http://www-128.ibm.com/developerworks/xml/library/x-injava/index.html.

38. Slominski, A., *XML Pull Parser*, http://www.extreme.indiana.edu/xgws/xsoap/xpp/xpp2/index.html.

39. Shrideep Pallickara and G. Fox, *On the Matching of Events in Distributed Brokering Systems.* . In Proceedings of the international Conference on information Technology: Coding and Computing (Itcc'04) Volume 2 - Volume 2 (April 05 - 07, 2004). ITCC. IEEE Computer Society, Washington, DC.

40. Pallickara, S., et al., *A Transport Framework for Distributed Brokering Systems*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. (PDPTA'03) Las Vegas June 2003.

41. Gunduz, G., S. Pallickara, and G. Fox, *A Framework for Aggregating Network Performance in Distributed Brokering Systems*, Proceedings of the 9th International Conference on Computer, Communication and Control Technologies.CCCT '03 July-August 2003.Orlando Florida.

42. Fox, G., S. Pallickara, and X. Rao, *Towards enabling peer-to-peer Grids*. 2005, Concurrency and Computation: Practice & Experience archive Volume 17 , Issue 7-8 (June 2005) Pages: 1109 - 1131. 2002 ACM Java Grande–ISCOPE Conference Part II.

43. Pallickara, S., et al., *A Security Framework for Distributed Brokering Systems*: Available from http://www.naradabrokering.org.

44. Majumdar, S., Eager, D. L., and Bunt, R. B. , *Scheduling in multiprogrammed parallel systems.* . SIGMETRICS Perform. Eval. Rev. 16, 1 (May. 1988), 104-113.

45. Kut, A. and D. Birant, *An Approach for Parallel Execution of Web Services* In Proceedings of the IEEE international Conference on Web Services (Icws'04) - Volume 00 (June 06 - 09, 2004). ICWS.

46. Kazi, I.H. and D.J. Lilja, *Coarse-Grained Thread Pipelining: A Speculative Parallel Execution Model for Shared-Memory Multiprocessors.* IEEE Trans. Parallel Distrib. Syst. 12, 9 (Sep. 2001), 952-966.

47. Francesco Pessolano and J.L.W. Kessels, *Asynchronous First-in First-out Queues* Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation (Springer-Verlag): p. 178-186.

48. P. Leach, M. Mealling, and R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace.* http://www.ietf.org/rfc/rfc4122.txt, July 2005.

49. Fox, G., Pallickara, S., and Parastatidis, S, *Toward Flexible Messaging for SOAP-Based Services.* In Proceedings of the 2004 ACM/IEEE Conference on

Supercomputing (November 06 - 12, 2004). Conference on High Performance Networking and Computing.

50. Arulanthu, A.B., et al., *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging* in Proceedings of the Middleware 2000 Conference, ACM/IFIP, Apr. 2000.

51. L. Bellissard, et al., *An Agent Platform for Reliable Asynchronous Distributed Programming* In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (October 18 - 21, 1999).

52. Langendoen K., R. Bhoedjang, and H. Bal, *Models for Asynchronous Message Handling* IEEE Parallel Distrib. Technol. 5, 2 (Apr. 1997), 28-38.

53. Buchmann, S.K.A., *Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services.* in Proceeding of the 28th International Conference on Very Large Data Bases; 2002

54. Thakur R., W. Gropp, and E. Lusk, *On implementing MPI-IO portably and with high performance.* In Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (Atlanta, Georgia, United States, May 05 - 05, 1999). IOPADS '99. ACM Press, New York, NY.

55. Amir Y., et al., *A cost-benefit flow control for reliable multicast and unicast in overlay networks.* IEEE/ACM Trans. Netw. 13, 5 (Oct. 2005), 1094-1106.

56. Shenker, S., *A theoretical analysis of feedback flow control.* In Proceedings of the ACM Symposium on Communications Architectures &Amp; Protocols (Philadelphia, Pennsylvania, United States, September 26 - 28, 1990). SIGCOMM '90. ACM Press, New York, NY.

57. Shivers, O., *Control flow analysis in scheme.* In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, United States, June 20 - 24, 1988). R. L. Wexelblat, Ed. PLDI '88. ACM Press, New York, NY.

58. Qiu D. and N.B. Shroff, *A new predictive flow control scheme for efficient network utilization and QoS.* In Proceedings of the 2001 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems (Cambridge, Massachusetts, United States). SIGMETRICS '01. ACM Press, New York, NY, 143-153.

59. Pallickara, S., *Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids.* Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003.

60. Fox, G., *2004. A Scheme for Reliable Delivery of Events in Distributed Middleware Systems.* In Proceedings of the First international Conference on Autonomic Computing (Icac'04) - Volume 00 (May 17 - 18, 2004). ICAC. IEEE Computer Society, Washington, DC, 328-329.

61. Tai, S., Thomas A. Mikalsen, and Isabelle Rouvellou, *Using Message-oriented Middleware for Reliable Web Services Messaging.* Lecture notes in computer science (Lect. notes comput. sci.) ISSN 0302-9743 , 2003.

62. S Maffeis and D.C. Schmidt, *Constructing Reliable Distributed Communication Systems with CORBA.* IEEE Comm., Feb. 1997.

63. P. T. Eugster, et al., *The many faces of publish/subscribe.* ACM Comput. Surv. 35, 2 (Jun. 2003), 114-131.

64. Aalst, W.M.P.v.d., *The application of petri nets to workflow management, J. Circuits Systems Comput. 8(1) (1998) 21-66.*

65. WFMC., *WorkflowManagement Coalition Terminology and Glossary(WFMC-TC-1011).* Technical report, Workflow Management Coalition,Brussels, 1996.

66. Ewa Deelman, et al., *GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists* hpdc, p. 225, 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02), 2002.

67. C. Berkley, et al., *Incorporating semantics in scientific workflow authoring* In Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM'05).

68. Oinn, T., et al., *Taverna: lessons in creating a workflow environment for the life sciences* Concurr. Comput. : Pract. Exper. 18, 10 (Aug. 2006), 1067-1100.

69. TIBCO Software Inc Inconcert, http://www.tibco.com.

70. Aggarwal , B.A., A. Chandra , M. Snir, *A model for hierarchical vmemory,* Proceedings of the nineteenth annual ACM conference on Theory of computing, p.305-314, January 1987, New York, New York, United States

71. IBM Lotus, http://www-306.ibm.com/software/lotus/.

72. Cao, J., et al., *GridFlow: workflow management for grid computing*, Proceedings of 3th International Symposium on Cluster Computing and the Grid, Tokyo, Japan, May 12-15, 2003. IEEE Computer Society Press, 198-205. p. 198-205.

73. von Laszewski, G., et al., *GridAnt–client-side workflow management with Ant,* Proceedings of 37th Hawaii International Conference on System Sceince, Island of Hawaii, Big Island, January 2004.

74. Curbera F, et al., *Business Process Execution Language for Web Services (BPEL4WS)* http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

75. Sriram Krishnan, P.W., and Gregor von Laszewski., *GSFL: A Workflow Framework for Grid Services.* In Preprint ANL/MCS-P980-0802, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, 1L 60439, U.S.A., 2002.

76. Web Service Choreography Interface (WSCI) 1.0 http://www.w3.org/TR/wsci/.

77. Angell, K.W., *Programmer's toolchest: Examining JPython: A Java test engine puts Python to the test*. p. 78.

78. Krishnan, S. and D. Gannon, *XCAT3: a framework for CCA components as OGSA services*, Proceedings of the 9th International Conferemce on Computational Science 2003 (HIPS 2003). IEEE Computer Society Press 90-97.

79. Frey, J., *Condor DAGMan: Handling Inter-Job Dependencies*. 2002, Technical report, University of Wisconsin, Dept. of Computer Science, http://www.cs.wisc.edu/condor/dagman.

80. Lorch, M. and D. Kafura, *Symphony–A Java-based Composition and Manipulation Framework for Computational Grids*. 2002, IEEE Computer Society Washington, DC, USA.

81. Thatte, S., *XLANG: Web Services for Business Process Design*. 2001.

82. Leymann, F., *Web Service Flow Language (WSFL)*, Technical report, IBM, May 2001, http://www-4. ibm. com/software/solutions/webservices/pdf/WSFL. pdf.

83. Erwin, D.W., *UNICORE—a Grid computing environment*. 2002, Springer. p. 1395-1410.

84. Hoheisel, A., *User tools and languages for graph-based Grid workflows.* Concurrency and Computation: Practice and Experience, 2005. 18(Special issue Workflow in Grid System): p. 1101-1113.

85. Petri, C.A., *Kommunikation mit Automaten*. 1962: Bonn.

86. Desel, J.a.J., Gabriel *"What Is a Petri Net? -- Informal Answers for the Informed Reader", Hartmut Ehrig et al. (Eds.): Unifying Petri Nets,* LNCS 2128, pp. 1-25, 2001.

87. H. A. James, K.A.H., and P. D. Coddington., *An Environment for Workflow Applications on Wide-Area Distributed Systems.* Technical Report DHPC-091, Distributed and High Performance Computing Group, Department of Computer Science, The University of Adelaide, May 2000. Submitted to HICSS'34.

88. J¨ungel, M., *E. Kindler, and M. Weber. The Petri Net Markup Language.* Petri Net Newsletter, 59:24–29, 2000.

89. J. Billington, e.a., *The Petri Net Markup Language: Concepts, Technology, and Tools.* Proc. Int'l Conf. Applications and Theory of Petri Nets 2003, W. van der Aalst and E. Best eds., LNCS, vol. 2679, Springer, 2003, pp. 483--505.

90. Aalst, W.M.P.v.d. and A. Kumar, *XML Based Schema Definition for Support of Inter-organizational Workflow.* University of Colorado and University of Eindhoven report, 2000.

91. XML Schema, http://www.w3.org/XML/Schema.html.

92. van Emde Boas, P., R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue*. 1976, Springer. p. 99-127.

93. Vuillemin, J. and U. de Paris-Sud, *A Data Structure for Manipulating Priority Queues*. 1978.

94. Handley, M., et al., *RFC3448: TCP Friendly Rate Control (TFRC): Protocol Specification*. 2003, RFC Editor United States.

95. Avid Karger , A.S., Andy Berkheimer , Bill Bogstad , Rizwan Dhanidina , Ken Iwamoto , Brian Kim , Luke Matkins , Yoav Yerushalmi, *Web caching with consistent hashing.* Proceeding of the eighth international conference on World Wide Web, p.1203-1213, May 1999, Toronto, Canada

96. Transmission Control Protocol (TCP) http://www.ietf.org/rfc/rfc793.txt.

97. David Karger, E.L., Tom Leighton, Matthew Levine, Daniel Lewin and Rina Panigrahy *Consistent hashing and random trees: Distributed cachine protocols for relieving hot spots on the World Wide Web* In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663 , 1997.

98. Abraham Silberschatz, G.G., Peter Baer Galvin, *Operating system concepts*. 2002: Addison-Wesley Reading, Mass.

99. Voelter, M., M. Kircher, and U. Zdun, *Patterns for asynchronous invocations in distributed object frameworks*, EuroPLoP 2003, http://www.kircher-schwanninger.de/michael/publications/AsynchronyEuroPLoP2003.pdf.

100. Tanenbaum, A.S. and M. Van Steen, *Distributed Systems: Principles and Paradigms*. 2001: Prentice Hall PTR Upper Saddle River, NJ, USA.

101. Laprie, J.C.C., A. Avizienis, and H. Kopetz, *Dependability: Basic Concepts and Terminology*. 1992, Springer-Verlag New York, Inc. Secaucus, NJ, USA.

102. Leymann, F. and W. Altenhuber, *Managing Business Processes an an Information Resource*. 1994. p. 326-348.

103. Hagen, C. and G. Alonso, *Exception handling in workflow management systems*. 2000. p. 943-958.

104. Gray, J., *Notes on Data Base Operating Systems*. 1978: Springer-Verlag London, UK.

105. Helal, A.A., A.A. Heddaya, and B.K. Bhargava, *Replication Techniques in Distributed Systems*. 1996: Kluwer Academic Pub.

106. von Neumann, J., *Probabilistic logics and the synthesis of reliable organisms from unreliable components*. 1956. p. 43-99.

107. Lee, P.A., et al., *Fault Tolerance: Principles and Practice*. 1990: Springer-Verlag New York, Inc. Secaucus, NJ, USA.

108. Borg, A., et al., *A Message System Supporting Fault Tolerance*, A.O.S. Review, Editor. 1983.

109. Ng, T.P. and S.S.B. Shi, *Replicated transactions*, in *IEEE 7th International Conference on Distriuted Computing Systems*. 1989. p. 474-480.

110. Shelley Zhuang, D.G., Ion Stoica, and Randy Katz, *On failure detection algorithms in overlay networks.* In INFOCOM'05, 2005.

111. Khoussainov, R. and A. Patel, *LAN security: problems and solutions for Ethernet networks*. 2000, Elsevier Science Publishers BV Amsterdam, The Netherlands, The Netherlands. p. 191-202.

112. Yajima, S., *Loosely coupled multiprocessor system capable of transferring a control signal set by the use of a common memory*. 1987, Google Patents.

113. Pena, C.J.C. and J. Evans, *Performance Evaluation of Software Virtual Private Networks (VPN)*. 2000. p. 522–523.

114. Park, M.H., et al., *Implementation and perforinance evaluation of hardware accelerated IPSec VPN for the home gateway*. 2005.

115. Pallickara, S. and G. Fox, *A scheme for reliable delivery of events in distributed middleware systems*, Proceedings of the IEEE International Conference on Autonomic Computing. ICAC'04 New York, NY. May 17-18 2004, pp. 328-329. p. 328-329.

116. Johnson, C.a.W., J *Future processors: flexible and modular.* In Proceedings of the 3rd IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis (Jersey City, NJ, USA, September 19 - 21, 2005). *CODES+ISSS '05. ACM Press, New York, NY, 4-6.* 2005

117. Majumdar, S., Eager, D. L., and Bunt, R. B. , *Scheduling in multiprogrammed parallel systems.* SIGMETRICS Perform. Eval. Rev. 16, 1 (May. 1988), 104-113. 1988.

118. Web Service Eventing (WS-Eventing), http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.

119. FINS, An Implementation of WS-Eventing, http://www.naradabrokering.org/FINS-Docs/.

Vitae

Name of Author: Beytullah Yildiz

Date of Birth: April 9, 1974

Place of Birth Izmit, TURKEY

Degrees Awarded:

July 2007 Ph.D. in Computer Science,
Indiana University
Bloomington, IN, U.S.A

May 2001 M.S. in Computer & Information Science
Syracuse University
Syracuse, NY, U.S.A

May 1997 B.S. in Electronics Engineering
Istanbul University
Istanbul, TURKEY